PCT/US97/22845 88256 S

Europäisches Patentamt	European Patent Office	Office européen des brevets

Bescheinigung	Certificate	Attestation

Die angehefteten Unterlagen stimmen mit der ursprünglich eingereichten Fassung der auf dem nächsten Blatt bezeichneten europäischen Patentanmeldung überein.

The attached documents are exact copies of the European patent application described on the following page, as originally filed.

Les documents fixés à cette attestation sont conformes à la version initialement déposée de la demande de brevet européen spécifiée à la page suivante.

Patentanmeldung Nr. | Patent application No. | Demande de brevet n°

96402785.8

Der. Präsident des Europäischen Patentamts:
Im Auftrag

For the President of the European Patent Office

Le Président de l'Office européen des brevets
p.o.

Hatten

LL.C. Hatten-Heckman

DEN [redacted]AAG,DEN
THE HAGUE, 23/10/97
LA HAYE,LE

Europäisches
Patentamt

European
Patent Office

Office européen
des brevets

Blatt 2 der Bescheinigung
Sheet 2 of the certificate
Page 2 de l'attestation

Anmeldung Nr.:
Application no.: 96402785.8
Demande n°:

Anmeldetag:
Date of filing: 18/12/96
Date de dépôt:

Anmelder:
Applicant(s):
Demandeur(s):

THOMSON multimedia
92050 Paris La Défense 5
FRANCE

Bezeichnung der Erfindung:
Title of the invention:
Titre de l'invention:

HD MPEG video recorder

In Anspruch genommene Prioriät(en) / Priority(ies) claimed / Priorité(s) revendiquée(s)

Staat: State: Pays:	Tag: Date: Date:	Aktenzeichen: File no. Numéro de dépôt:

Internationale Patentklassifikation:
International Patent classification:
Classification internationale des brevets:

H04N7/00

Am Anmeldetag benannte Vertragstaaten:
Contracting states designated at date of filing: AT/BE/CH/DE/DK/ES/FI/FR/GB/GR/IE/IT/LI/LU/MC/NL/PT/SE
Etats contractants désignés lors du depôt:

Bemerkungen:
Remarks:
Remarques:

APPENDIX AA

DETAILED DESCRIPTIONS

INVENTION DISCLOSURE

This disclosure covers one of the changes made to the original HDMPEG memory compression scheme disclosed in RCA # 87,791, in order to simplify the hardware implementation. Specifically, this disclosure covers the encoding of overhead information in each block of compressed data.

The 8x8/4x4 block of luma/chroma data to be compressed is scanned to determine the minimum and maximum values as before. The min and max values are then quantized, giving a discrete minimum and range value. The range determines the quantizer used on the data in the current luma/chroma block. As shown in the table below, seven ranges are allowed.

The compression algorithm used is a modified DPCM loop. One key item to ensure good compression is the accuracy of the first pixel, since the predictor for every other pixel in the block is dependent on this value. In the HD-MPEG IC, the first pixel is treated as a special case and encoded in the overhead bits at the beginning of each block of compressed data.

From Appendix G of the HDMPEG spec:

Each block of compressed data begins with overhead bits to indicate the parameters necessary for decompression. Three types of information may be included: the range of the data, the minimum value of the data block, and a representation of the first pixel in the block. For chroma blocks, the header bits include a 3 bit range, and the first pixel truncated by 1 bit or fewer, if the resulting number of overhead bits would be less than 8. Luma blocks include the three bit range, a quantized representation of the minimum value of a block, and a representation of the first pixel. In most cases, this representation is the first pixel value minus the block minimum truncated. The overhead bits are defined below in the table below. In the case of Range=256, the first pel is simply quantized to 7 bits.

INVENTION DISCLOSURE

	Range	0	1	2	3	4	5	6	7	8	9	10	11	12	13	comment	bits lost to overhead
Chroma	16	0	0	0	P_7	P_6	P_5	P_4	P_3								3
	32	0	1	0	P_7	P_6	P_5	P_4	P_3							first pel trunc to 4	3
	64	0	1	1	P_7	P_6	P_5	P_4	P_3							first pel trunc to 5	4
	96	1	0	0	P_7	P_6	P_5	P_4	P_3	P_2						first pel trunc to 6	5
	128	1	1	0	P_7	P_6	P_5	P_4	P_3	P_2						first pel trunc to 6	5
	192	1	0	1	P_7	P_6	P_5	P_4	P_3	P_2	P_1					first pel trunc to 7	6
	256	1	1	1	P_7	P_6	P_5	P_4	P_3	P_2	P_1					first pel trunc to 7	6
		↑ range ↑			↑ First Byte ↑					↑ Second Byte ↑							
Luma	16	0	0	0	D_7	D_6	D_5	M_7	M_6	M_5	M_4	M_3	M_2	M_1	M_0		10
	32	0	1	0	D_7	D_6	D_5	D_4	M_7	M_6	M_5						6
	64	0	1	1	D_7	D_6	D_5	D_4	D_3	M_7	M_6	M_5					7
	96	1	0	0	D_7	D_6	D_5	D_4	D_3	D_2	M_7	M_6	M_5				8
	128	1	1	0	D_7	D_6	D_5	D_4	D_3	D_2	M_7	M_6	M_5				8
	192	1	0	1	D_7	D_6	D_5	D_4	D_3	D_2	M_7	M_6	M_5				9
	256	1	1	1	P_7	P_6	P_5	P_4	P_3	P_2	P_1					first pel trunc to 7	6
		↑ range ↑			↑ First Byte ↑					↑ Second Byte ↑							

▨	range bits
M	minimum value
P	First pel, quantized
D	First pel - min, unless noted otherwise

Note: the last column gives the number of extra bits used in coding the first pixel. In the M/2 mode, which gives 50% compression, it is expected to use 4 bits per pixel. This number indicated the number of bits over 4 used for the first pixel.

Table G3- Overhead Bits

Range	0	1	2	3	4	5	6	7	8	9	10	11	12	13	comment	bits lost to overhead
16	0	0	1	D_3	D_2	D_1	M_7	M_5	M_5	M_4	M_3	M_2	M_1	M_0	Δ & mimum value	10
32	0	1	0	D_4	D_3	D_2	D_1	I_2	I_1	I_0					Δ & mimum index	6
64	0	1	1	D_5	D_4	D_3	D_2	D_1	I_2	I_1	I_0				Δ & mimum index	7
96	1	0	0	D_5	D_5	D_4	D_3	D_2	D_1	I_2	I_1	I_0			Δ & mimum index	8
128	1	1	0	D_6	D_5	D_4	D_3	D_2	D_1	I_2	I_1	I_0			Δ & mimum index	8
192	1	0	1	D_6	D_5	D_4	D_3	D_2	D_1	I_2	I_1	I_0			Δ & mimum index	9
256	1	1	1	P_7	P_6	P_5	P_4	P_3	P_2	P_1					first pel trunc to 7	6
	↑ range ↑			↑ First Byte ↑					↑ Second Byte ↑							

▨	range bits
M	minimum value
I	index into Minimum Lookup Table
P	first pel, quantized
D	first pel - min, unless noted otherwise

The loop glue section checks for quantization induced error in the compression process. If *e* is negative, the Range maximum value is added to delta before it is encoded. At decompression, when this value is added to the predictor, it will be out of range, the Range Max value will be subtracted, and the correct result is obtained. An error could be introduced in this method by the quantizer: if a negative value is decreased by quantization such that it no longer goes out of range when added to Pred or if a positive value is increased such that it goes out of range. This can be checked using the following:

$$e' + PRED \geq R$$

If we notice that *R* and *PRED* are available earlier in the loop than *e'* and that this equation requires two operations after *e'* is available, we can use

$$e' \geq R - PRED$$

For negative values of *e*, the tables subtract *R* from the actual output to produce $e_m' = e' - R$.

$$e'_m + PRED \geq 0$$

Delta	Quant ≥ (Range - Pred)	$\text{Quant}_{max} + P \geq 0$	Use
pos	0	X	QUANT
pos	1	X	ALT
neg	X	0	ALT-MAX
neg	X	1	QUANT-MAX

G2.4.8. Compression Controller

G2.4.8.1. Overview

Vertical Sample Rate Conversion

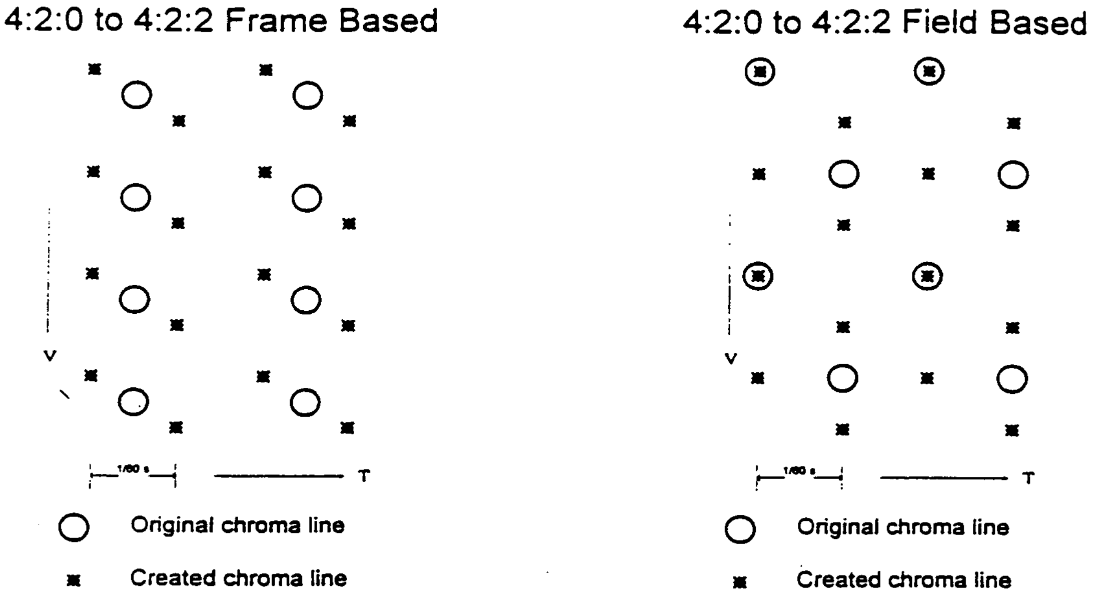
The display section of a high definition MPEG decoder must be able to convert the incoming decompressed video to a format compatible with the display used in the particular television chassis. Likely vertical output formats include: 1080i (1080 active line - interlaced), 480p (480 active line - progressive), and 480i (480 active line - interlaced). Likely vertical input formats include: 240i, 360p, 480p, 480i, 720p, and 1080i. The luma vertical sample rate converter (luma V-SRC) is designed to convert any of these likely inputs to the desired output by using a generalized design. This design allows the possibility of supporting other format conversions without hardware change. The filter coefficients and the sequence through them (sequencing through the phases of the sub-filters) are controlled through the host bus (by the microprocessor).

The vertical sample rate converter for luma will be a 3 tap polyphase filter type. The implementation, shown in Figure 3, uses a single delay line and uses recursive feedback and variable filter coefficients the implement this 3 tap polyphase filter. Care is also taken to process data through the vertical filter only when it is needed by the horizontal sample rate converters, thus eliminating the need for FIFOs between the vertical and horizontal sample rate converters.

In addition, regardless of input luma format, the input format has a 4:2:0 relationship between luma and chroma. Hence, in addition to the conversions for luma, a 4:2:0 to 4:2:2 conversion must also be done for chroma by the chroma vertical sample rate converter (chroma V-SRC). In the case of chroma, a 2 tap filter is used for the combined vertical resampling and 4:2:0 to 4:2:2 conversion.

Conversion from 4:2:0 to 4:2:2 Format

In all cases some processing of chroma is required as video is stored in memory in 4:2:0 format and the display device expects 4:2:2 format data. Usually this chroma processing will be included with any other required vertical processing. The figure below illustrates the vertical/temporal relationship of input and output chroma lines when 4:2:0 to 4:2:2 only conversion is required (i.e. receive 480 interlace and display 480 interlace or receive 1080 interlace and display 1080 interlace).


4:2:0 to 4:2:2 Frame Based

4:2:0 to 4:2:2 Field Based

Figure 1.
The frame based case is used when the MPEG picture header indicates the 4:2:2 to 4:2:0 process was frame based. In this case all the chroma lines are used to generate the first or top field then again to generate the second or bottom field.

The field based case is used when the MPEG picture header indicates the 4:2:2 to 4:2:0 process was field based. In this case the even chroma lines (starting with 0) are used to generate the first or top field; the odd chroma lines are used to generate the second or bottom field.

Conversion from 720 Progressive to 1080 Interlace

The figure below illustrates the vertical/temporal relationship of input and output luma and chroma lines when the 720 progressive format is converted to 1080 interlace. The chroma case includes the 4:2:0 to 4:2:2 conversion.

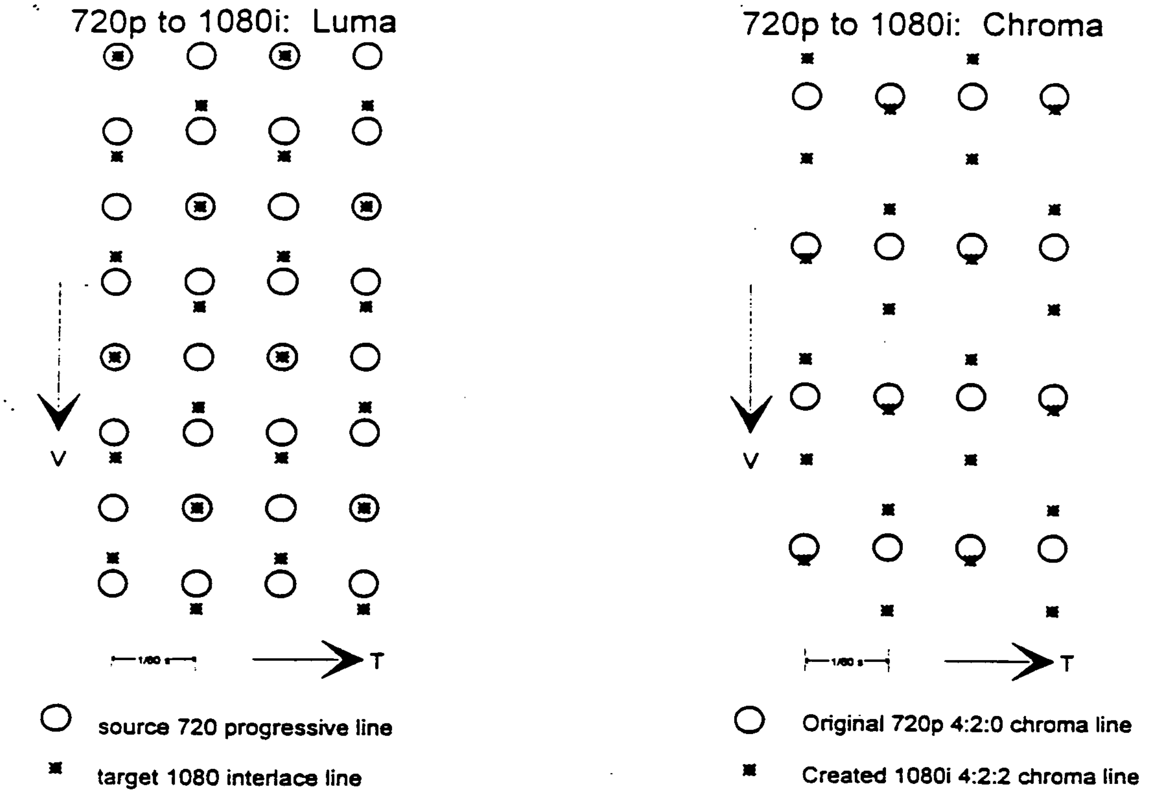
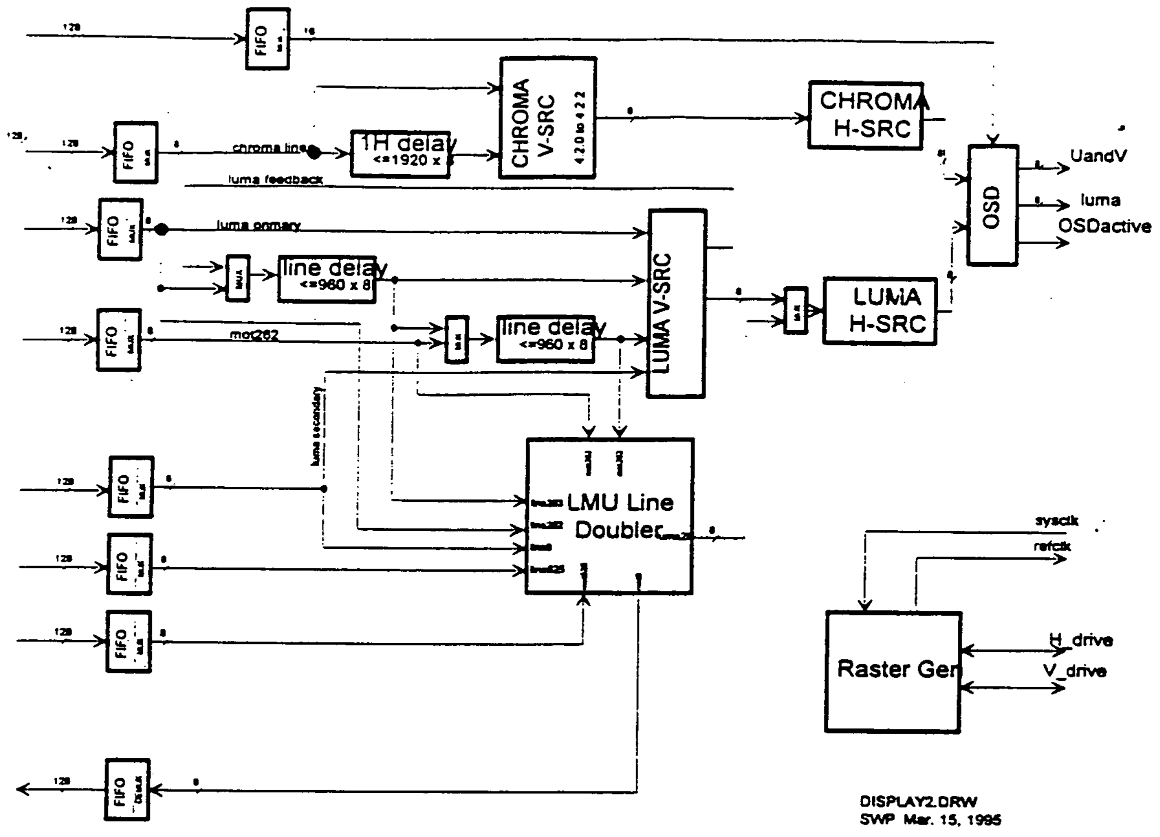

720p to 1080i: Luma
V
1/60 s
T
source 720 progressive line
target 1080 interlace line
720p to 1080i: Chroma
V
1/60 s
T
Original 720p 4:2:0 chroma line
Created 1080i 4:2:2 chroma line

Figure 2.
Both the luma and chroma processing occurs only in the vertical direction. No temporal processing is used.



DISPLAY2.DRW
SWP Mar. 15, 1995

Figure 3. Implementation of display section for high definition MPEG decoder.

Horizontal Sample Rate Conversion

The horizontal sample rate converter (H - SRC) must be able to convert the horizontal format (in terms of number of active pixels) to that desired for the output. The actual H - SRC filter always does an upconversion; that is the mber of pixels on the output is always greater than or equal to the number of pixels at the input. The filter is realized with a fixed response, 5-tap pre-filter, and a 4-tap, 16 times oversampling filter. The filter structure for multiplexed chroma (Cr,Cb pairs) is essentially the same as for luma. The entire display runs on a clock synchronous with the output. The control logic generates a shift enable whenever the shift registers need new data. This shift enable also results in data flowing from the FIFO's through the vertical SRC to the horizontal filters. This eliminates the need for a FIFO between the vertical and horizontal filter sections.

Progressive Scan Conversion

For the special case of 480i to 480p conversion, the V - SRC is not used. Rather, a progressive scan conversion algorithm (called the LMU with film mode) is used. This algorithm estimates motion in the image on a pixel by pixel basis and chooses the best combination of spatial and temporal (adjacent field information) to calculate the pixels for the lines not originally present in the field currently displayed. This results in a superior image relative to that produced by vertical filtering each field.

INVENTION DISCLOSURE

The following terms will be used:

Decoder: the section that converts the encoded, variable length data to a quantized value that will be supplied to the predictor in the decompressor.

Decompressor: the DPCM loop that uncompresses the quantized data.

Wai-man's original adaptive quantization caused an implementation problem in designing the decoder/decompressor. The compression uses a variable length code. In the M/2 mode, where 50% compression is desired (4 bits per pixel), each pixel is encoded using a 3 bit, 4 bit, or 5 bit code word. Frequently occurring values are encoded with 3 bits, which produces a bit savings. If enough bits are saved to compensate for the overhead bits in each data block, the 5 bit table can be used producing less quantization error. The 3 bit table is also used if there has not been enough bit savings to meet the target compression.

To decode a variable length data block in hardware, a "barrel-shifter" circuit is used to discard bits as they are decoded. The hardware in the HDMPEG IC has two requirements: it must be as small as possible to minimize IC cost. This is accomplished by designing the barrel shifter to shift as few bits at a time as possible. The second requirement is that data must be decompressed continually, with few wasted clocks per block of data. For the HDMPEG IC, the barrel shifter is able to shift 6 bits at a time during normal operation.

The problem with the original algorithm occurs for a block which contains low detail, such as a block with uniform intensity, which could be compressed to a size smaller than the target compression. Whenever the bit savings in a given block is positive, 5 bit code words are used to improve quantization. If, however, a value corresponding to a 3 bit code word keeps occurring, the bit savings will continue to increase. If there is a bit savings when the compression is complete, the block would be padded with 0's because all compressed blocks must be the same size.

To demonstrate the problem, consider a low detail luma block: its 64 pixels could conceivably be compressed to 50 three bit words and 14 four bit words. A 50% compressed luma block should contain (64 * 4) or 256 bits. This example block would contain 150 + 56 + 6 (overhead bits) = 212 bits. The decoder will shift off bits as they are used, so that when the block is complete and 64 pixels are produced, only 212 bits will have been shifted out of the decoder. This leaves 44 bits in the decoder. To clear this in few clock cycles, the barrel shifter would need to shift many more bits at a time which would increase the size. To minimize the size, it would require many clock cycles to shift these padded bits out of the decoder.

The solution is to prevent a compressed block from ever becoming smaller than the target compression. A luma block of data is shown in Table 1, where the pixels are numbered in the order they appear in a block of data. In 50% (4 bit/pixel) mode, each pixel can be coded with 3, 4 or 5 bits and either use one extra bit, use no extra bits, or save one bit. Table 2 shows at each pixel location, the maximum number of extra bits that could be used at that point in the block. For example, when the pixel #64 is being coded, only one extra bit is useful, since it is the last pixel of the block and it could be coded with 5 bits. Any additional bits that were saved in this block are wasted and require padded data to be added at the end of the block.

To prevent this, the encoder and decoder will compare the bit savings for the block to the values in table 2. If the bit savings is ever greater than or equal to the value in Table 2 and a 3 bit word is requested, a 5 bit code word will be used instead. This guarantees that every compressed data block will be the same size without adding any padded data at the end. This will not increase the quantization table size in the compressor or decompressor. The compressor will choose the 3 bit code word as in the present algorithm, but it will shift out 5 bits, the 3 bit code word plus 2 padded bits. The decompressor will do the same, it will decode the original 3 bit code word, detect the condition by comparing bit savings to Table 2, and shift out the next 5 bits. This is in effect dispersing the padding bits throughout the block instead of placing them at the end of the block.

Table 1- Pixel order in luma block

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

Table 2- Maximum savings needed at each location

64	63	62	61	60	59	58	57
56	55	54	53	52	51	50	49
48	47	46	45	44	43	42	41
40	39	38	37	36	35	34	33
32	31	30	29	28	27	26	25
24	23	22	21	20	19	18	17
16	15	14	13	12	11	10	9
8	7	6	5	4	3	2	1

INVENTION DISCLOSURE

Patent Disclosure: Interleaved Pipelined DPCM Chroma/Luma Predictor for Video Processing

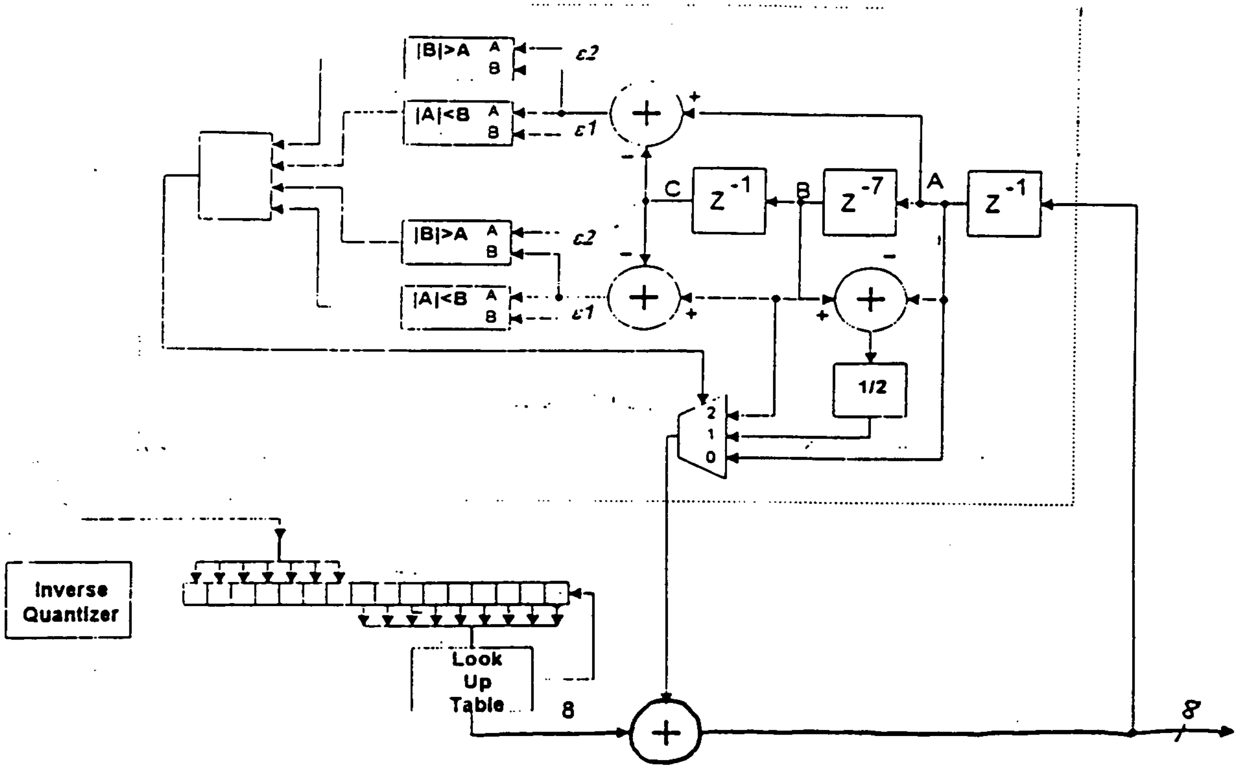
Problem: Compressing and decompressing video using differential pulse code modulation (DPCM) is very bandwidth intensive yet very serial based processing. A video predictor requires a fair amount of hardware and contains multi-path feedback which makes it very speed critical for each cycle. The serial processing also makes high throughput difficult. The obvious choice is to use parallel paths to process the data but this does not address the speed problems of the feedback loop. The speed of the video processing is a limitation in today's technologies. Due to technology limits, the processing must run at a slower clock rates which means a larger number of processors to meet the desired bandwidth. All of this means more die size and cost in processing the data. In summary, the way to implement a differential pulse code modulation (DPCM) system for compressing and decompressing video data is not obvious nor documented.

Solution: By interleaving multiple blocks of data at one time, the speed requirements for any single operation can be slowed to reasonable rates, the direct algorithm can be efficiently implemented in hardware, and the feedback loops can now be pipelined which was not possible as a single block. The technique described below addresses these tradeoffs to make a processor that allows a choice of old or new technologies without a great increase in hardware to meet the same bandwidth requirements. Interleaving also allows the system to be development without an interleave concept while still using the interleaving internal to the blocks.

Figı 1 shows the classic style of predictor loop. This is documented in Anil Jain's book Fundamentals of Digital Image Processing, r ;a84.

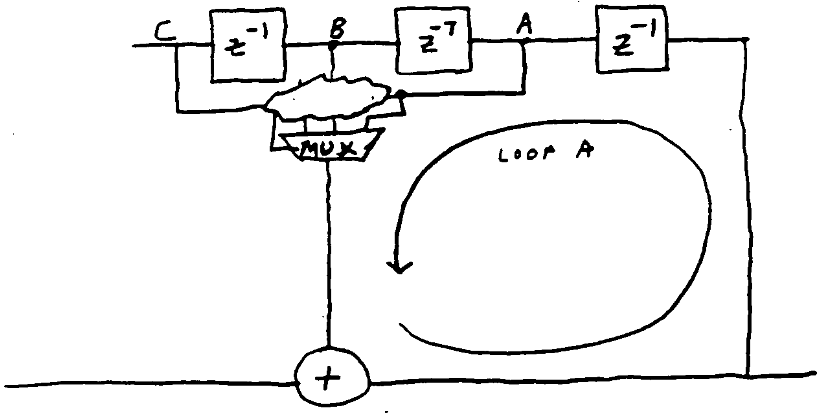
The remaining figures show the development of the pipelined approach for the same system. The pipeline concept is based on using additional matched delays to achieve a function in parallel that originally was performed in series. When multiple feedback paths and adjacent pixel dependencies are found, the sum of all pipelined delays must be zero in any path or the original algorithm would be lost. The system requirements would actually need a negative delay to compensate for a positive delay in the feedback path which is not possible in a single block path. If two blocks are processed and shifted together, the clock doubled, the system does not change in bandwidth but now the resolution of the clock versus data is 2 times the previous system. We are now capable of shifting the data delays around to provide delays that look negative or positive to the feedback loop so no change in the original algorithm has been made.

The final result is a system that can be broken up with delays to minimize speed/technology issues, high data rates, and a cost effect IC structure that minimizes the hardware. This is also a very versatile structure in the manner in which it is implemented since both chroma and luma data can flow through the block on opposite phases of the clock using only a simple control. The interleaving technique is used in additional blocks of the system to help process the signal with speed sensitive technologies.

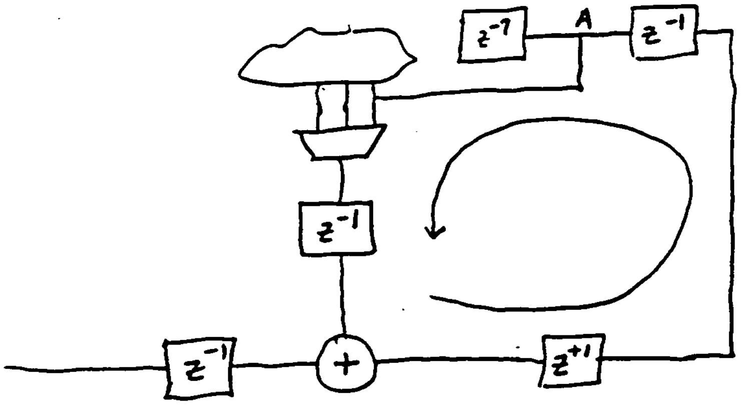


BASIC DECOMPRESSION FUNCTION IN ITS SIMPLIST FORM.

THE DELAY LINE IS NEEDED SINCE WE MUST CAREFULLY WATCH ALL BOUNDARY CONDITIONS WHEN DOING THE 2 DIMENSIONAL DCPM LOO. THE DELAY LINES HELP ORGANIZE THE DATA FOR PROCESSING BUT HE COMPARATORS AND ADDITIONAL LOGIC MAKE THE DECISIONS CONCERNING THE PREDICTOR OUTPUTS.

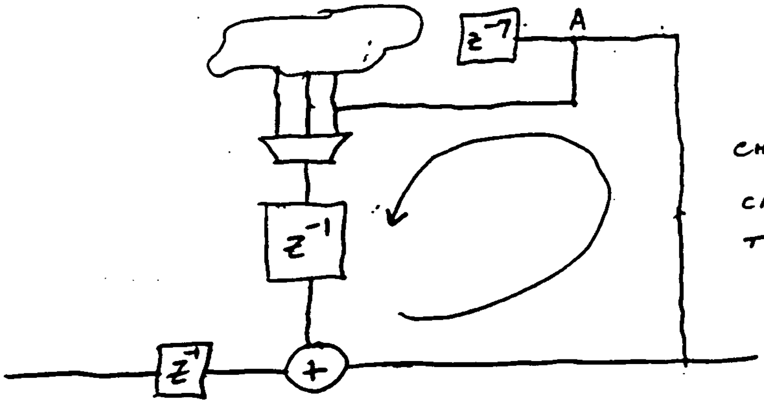


ONE OF MANY FEEDB, PATHS FOR THE PREDICTOR AND DECOMPRESSION CIRCUI

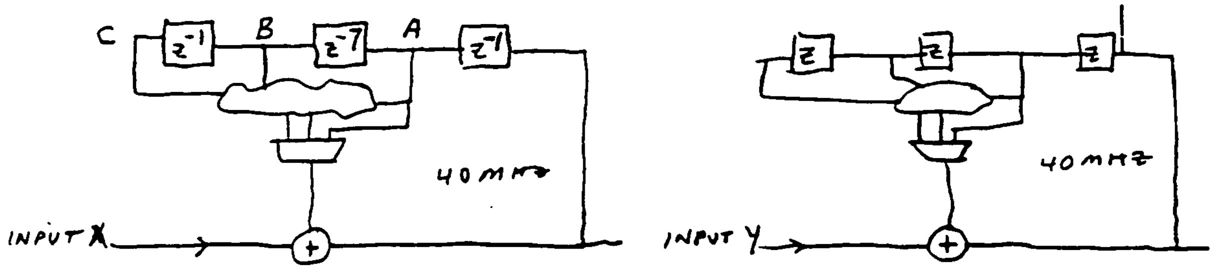


DELAYS ARE EQUAL IN LOOP SINCE $z^{-1} + z^{+1} + z$

WE CAN NOT MAKE THE z^{+1} ON THE IC.



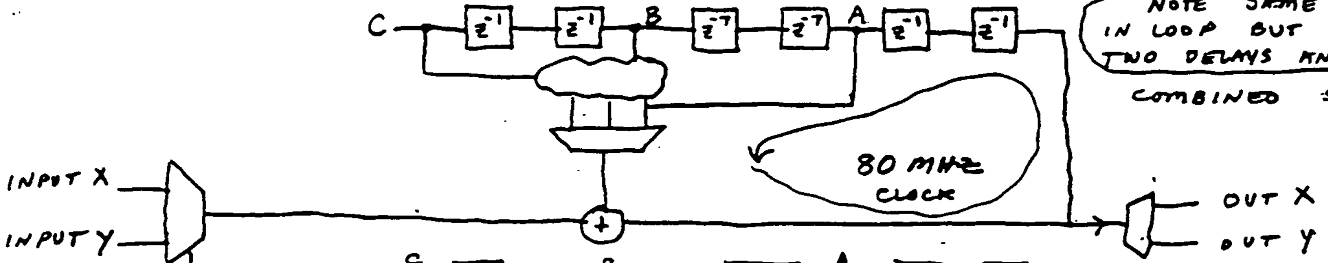
DELAYS SHIFTED TO CHANGE SYSTEM BUT CRITICAL LOOP REMAINS THE SAME.



WE NEED ONE DECOMPRESSOR AT 80MHZ OPERATION. IC TECHNOLOGY WILL ONLY SUPPORT <40MHZ DUE TO THE FEEDBACK LOOPS.

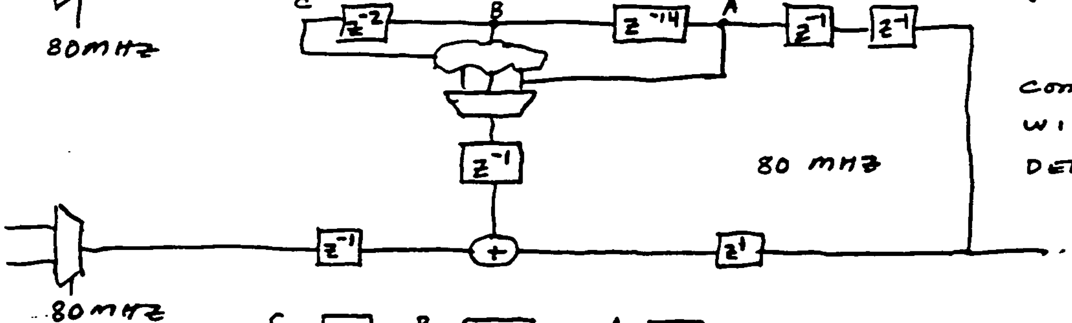
TWO DECOMPRESSIIN ARE NEEDED AT 40 MHZ TO GIVE 80MHZ DATARATES.

COMBINE THE TWO SYSTEMS

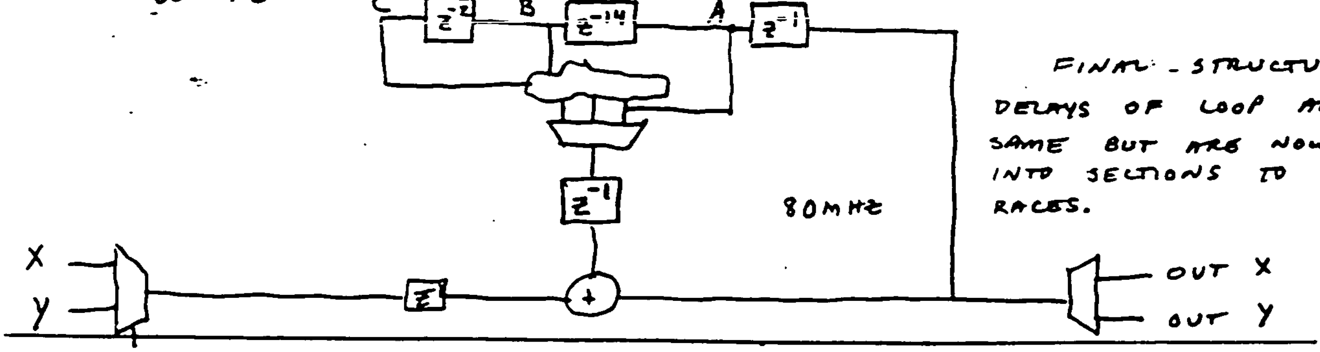


NOTE SAME DELAY FOUND IN LOOP BUT NOW USING TWO DELAYS AND 2X CLOCK

COMBINED STRUCTURE



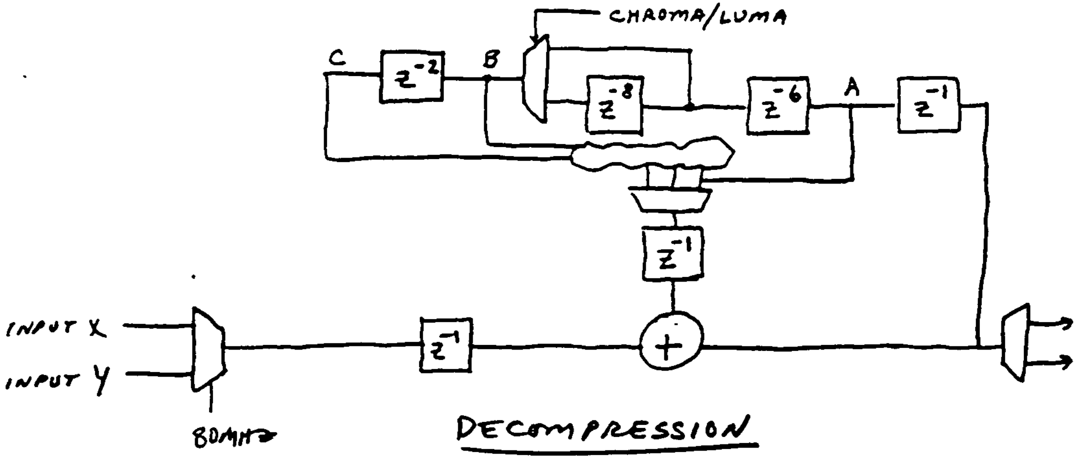
COMBINED STRUCTURE WITH PIPELINED DELAYS ADDED



FINAL - STRUCTURE

DELAYS OF LOOP ARE THE SAME BUT ARE NOW BROKEN INTO SECTIONS TO CONTROL RACES.

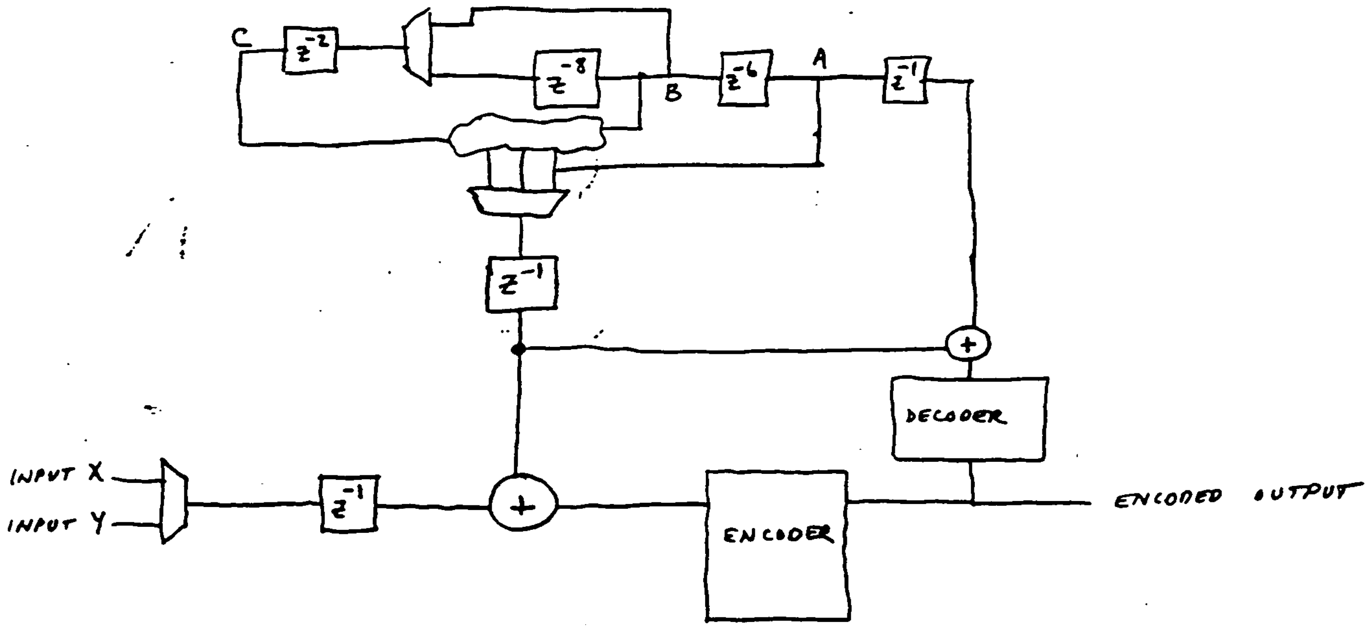
THE STRUCTURE SHOWN WAS USED FOR LUMA PROCESSING. NOW I WILL ADD CHROMA/LUMA PROCESSING WITH ONE PIECE OF HARDWARE.



DELAYLINE CHANG
WITH BLOCK TYPE.
THAT THIS STRUCT
CAN DO BOTH O
LUMA AND ONE C
BLOCK AT THE SA
TIME.

DECOMPRESSION

ADDITIONAL BENEFITS ARE FOUND IN THE COMPRESSOR SINCE THE LOOPS ARE GREATER AND THE PROCESSING IS MORE COMPLEX.



INVENTION DISCLOSURE

This disclosure discusses the hardware implementation of the memory decompression used in the HDMPEG IC.

The HDMPEG memory decompression is a DPCM loop (Figure 1) with modifications as specified in RCA # 87,791. It is important to note there is a loop that causes a given pixel's compressed value to be dependent on the results from the pixel to its immediate left, above it, and on its upper left diagonal. This is a serial operation that requires each pixel to be calculated in one clock cycle.

The decompression is placed between the local memory and the Motion Compensation Unit and between the local memory and the display section (figure 2). The decompression must, therefore, keep up with the data read from memory. In the highest resolution, 1920 x 1088, this requires:

$$1920\cdot 1088\cdot 30\cdot (8+4)/8\cdot 6 = 564{,}019{,}200\,{}^{byte}\!/_{sec}$$

$$\mathrm{Hor}\cdot \mathrm{Vert}\cdot \mathrm{fps}\cdot (\mathrm{Y}+\mathrm{C})\mathrm{bits}/\frac{\mathrm{bits}}{\mathrm{byte}}\cdot\text{" } predictor\_factor\text{"}$$

The *"predictor_factor"* is arrived as follows: each macroblock of data from the pipe could require 4 predictors. Due to memory compression inefficiency (the data in memory is "quantized" to 8x8 blocks, so an individual line of luma data cannot be accessed), 6 times as much data must be retrieved from memory.

The decompression section uses an 81 MHz clock, and therefore, a single decompressor section will produce 81,000,000 *byte/sec* multiplied by the inefficiency of the compression, 16/17 chroma 64/65 luma. A macroblock contains 4/6 luma data and 2/6 chroma data. This gives an average capacity of a single decompressor of 78,580,995 *bytes/sec.* Eight (actually 7.1) decompressors are required for the motion compensation section of the HDMPEG IC. To facilitate data flow and allow an individual to always decompress either luma or chroma, nine decompression units are used.

The display section requirements can be calculated similarly, except no *"predictor_factor"* is required. Two decompressors are needed for the display section.

A second problem encountered involved operating the DPCM decompression loop at the higher clock rate of 81 MHz. This frequency was chosen because it was already required by the display section. It was estimated that if the DPCM loop was pipelined and two independent blocks were interleaved on alternate clocks, the decompressors could operate at this frequency. This provides two benefits: there are now two clock cycles to perform the DPCM operations (from the pipelining), and there is no throughput penalty (due to interleaving data).

By pipelining, a decompressor produces 128 compressed bytes of data in 130 clock cycles (there is a luma cycle lost to overhead operation) which is equivalent to one non pipelined compressor. This aspect of the design is detail in a disclosure RCA ??? by Kranawetter and Schultz.

The final architecture is shown in figure 2. The nine decompression sections feed uncompressed data to the motion compensation unit and two decompression sections provide data to the display section.

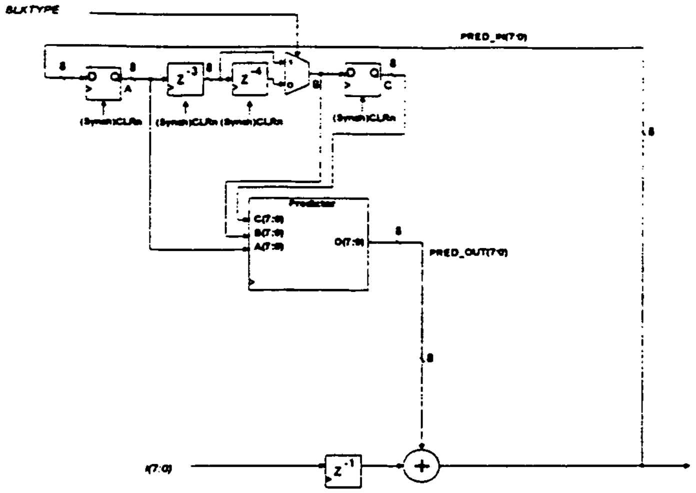


Figure 1

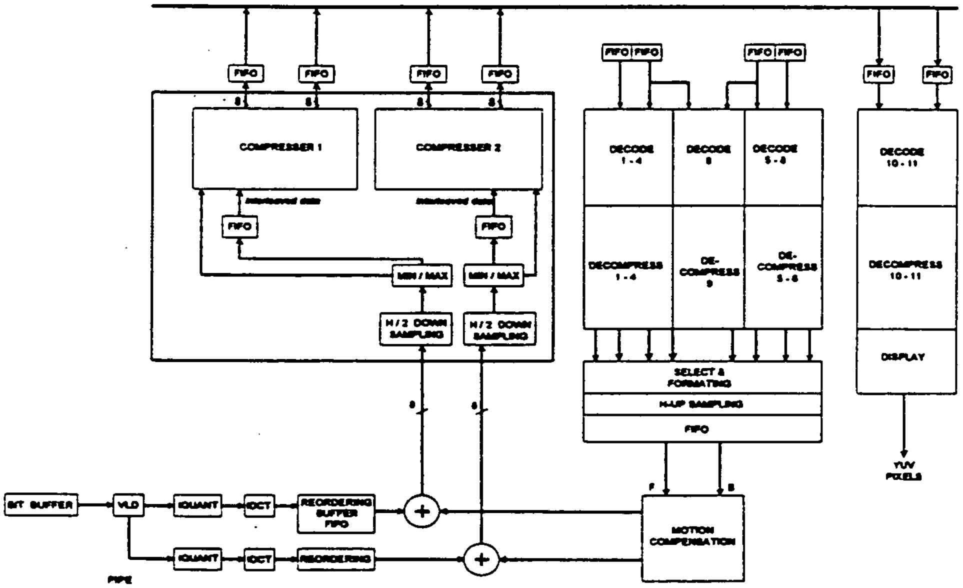


Figure 2

INVENTION DISCLOSURE

This disclosure discusses the hardware implementation of the memory compression used in the HDMPEG IC.

The HDMPEG memory compression is a DPCM loop (Figure 1) with modifications as specified in RCA # 87,791. It is important to note there is a loop that causes a given pixel's compressed value to be dependent on the results from the pixel to its immediate left, above it, and on its upper left diagonal. This is a serial operation that requires each pixel to be calculated in one clock cycle.

The compression is placed between the pipe output and the local memory (figure 2). The compression must, therefore, keep up with the data produced by the pipe. In the highest resolution, 1920 x 1088, this requires:

$$1920 \cdot 1088 \cdot 30 \cdot (8+4)/8 = 94{,}003{,}200\,{}^{byte}\!/_{sec}$$

$$\mathrm{Hor} \cdot \mathrm{Vert} \cdot \mathrm{fps} \cdot (\mathrm{Y} + \mathrm{C})\mathrm{bits} / \frac{\mathrm{bits}}{\mathrm{byte}}$$

The compression section must use a 54 MHz clock, and therefore, a single compressor section will produce 54,000,000 *byte/sec* multiplied by the inefficiency of the compression, 16/17 chroma 64/65 luma. A macroblock contains 4/6 luma data and 2/6 chroma data. This gives an average capacity of a single compressor of 52387330 *bytes/sec*. Two compressors are required for the HDMPEG IC.

A second problem encountered involves the sheer number of operations that must be executed in a single cycle of the compression DPCM loop: two adds, a comparison, several multiplexers, a quantization, and an overrun check. To enable this timing to be met, the loop must be pipe-lined giving an extra clock cycle to complete the operations. In order not to lose throughput due to the pipelining, two independent blocks of data are interleaved on alternate clocks so that a pixel is always being generated. This has the net effect of producing 128 compressed bytes of data in 130 clock cycles which is equivalent to one non pipelined compressor. This aspect of the design is detail in a disclosure RCA ??? by Kranawetter and Schultz.

The final architecture is shown in figure 2. The two MPEG pipes feed interleaved blocks of data to the two compression sections. The interleaved data passes through the H/2 block and the min/max scan sections and into a FIFO. Once an entire block has completed the Min/Max scan, it is fed to the compressor, which produces two outputs for the interleaved compressed blocks of data.

INVENTION DISCLOSURE

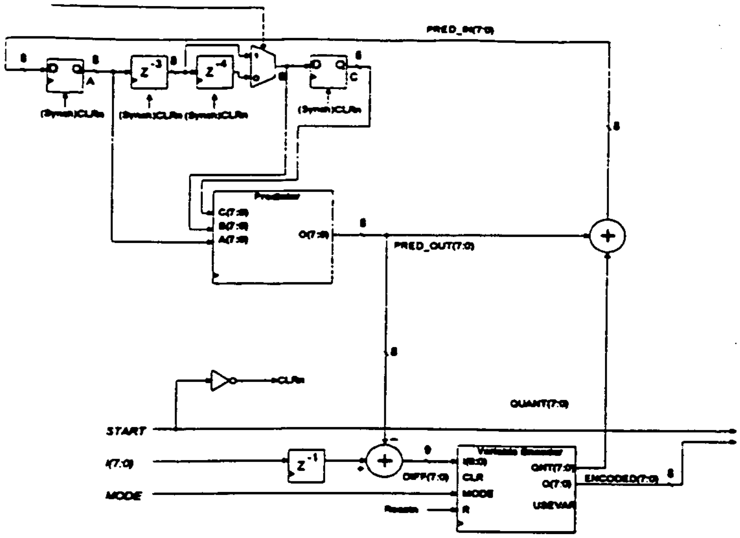


Figure 1

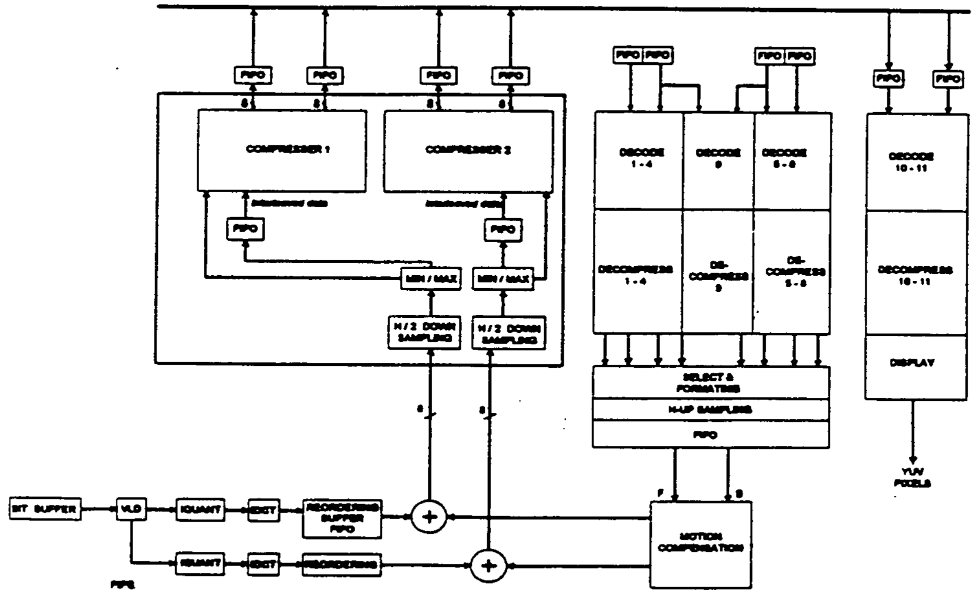


Figure 2

INVENTION DISCLOSURE

This disclosure discusses the hardware implementation of the memory compression/decompression used in the HDMPEG IC.

PRIOR ART:

In traditional NTSC digital video processing, composite video is frequently partitioned into a luminance component and multiplexed UV. An analog signal is sampled with a 4FC clock and combed. Since the chroma is much lower bandwidth, it can be sampled at half the rate of luma. The two components would look as follows:

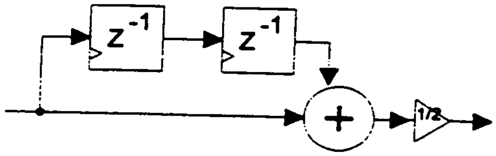
Luma:	$Y_0\, Y_1\, Y_2 \ldots$
Chroma:	$U_0\, V_0\, U_1\, V_1 \ldots$

If processing such as filtering is performed on the multiplexed chroma, it is possible to take advantage of the interleaving of the two chroma components. For example, if the chroma needs to be filtered with the filter

$x_n(1+Z^{-1})$,

the desired output of the filter would be ½ (U_1+U_0), ½ (V_1+V_0), ½ (U_2+U_1), ½ (V_2+V_1), ...

simple circuit to accomplish this is:



This circuit performs the needed function on both components while only using one adder.

As stated in RCA 88,228, there are several advantages in interleaving the operation of the memory reduction sections of the HDMPEG IC. It allows multiple clock cycles to perform operations and thus allow more operations to be performed at a given clock rate without impacting the effective throughput of the system.

A problem encountered in the memory reduction, however, was the presence of state machines that control the compress/decompress processes. A state machine relies on its current state and the values of certain signals and busses from the circuit to determine its next state. It is typically implemented with registers, which store the current state of the state machine, and combinatorial logic that uses the register outputs and additional signals from the circuit being controlled to generate the next state. In an interleaved system, the status of the various nets corresponds to two independent input streams on alternating clocks.

The obvious solution would be to have two state machines controlling the two interleaved bitstreams. By applying the same interleaving principles to the state machine, the design is simplified, timing is helped, and area is saved. Now, the state machine registers and additional signals from the circuit both correspond to two independent input streams in alternating clocks.

INVENTION DISCLOSURE

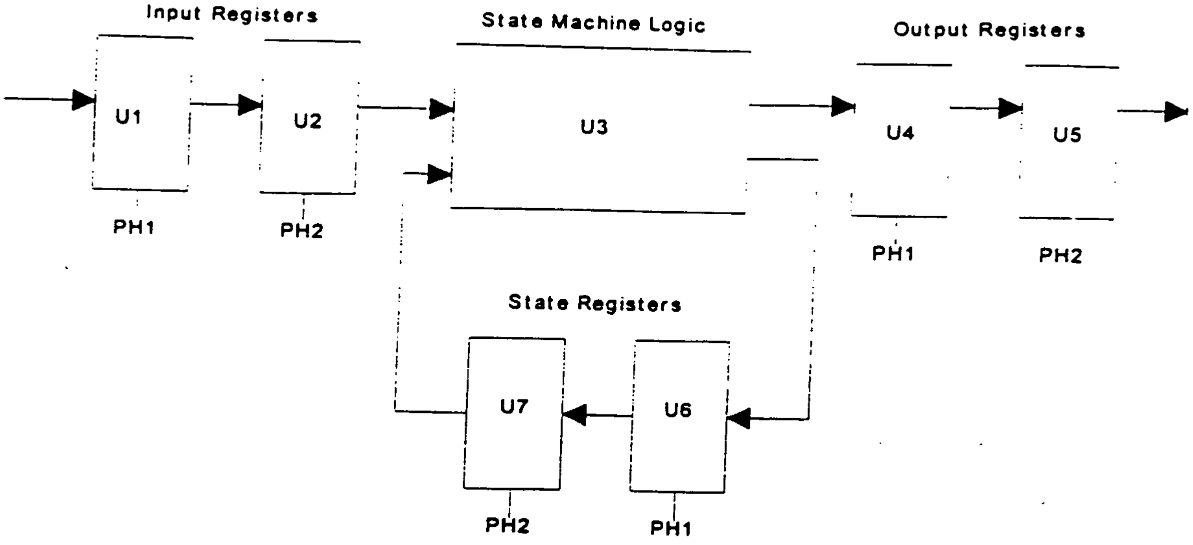


Fig 1

Shown in Fig. 1 is the embodiment of the invention. U1-U2 store a set of input variables. U1 is clocked by phase one of the master clock. U2 is clocked by phase 2 of the master clock. PH1 represents phase one and PH2 represents phase two. The master clock is twice the frequency of PH1 and PH2. Every two master clocks PH1 pulses high and then PH2 pulses high. Therefore PH1 and PH2 are non-overlapping clocks.

PH1 latches data into the registers U1,U6, and U4. PH2 latches data into registers U2,U7, and U5. PH1 represents one phase of a state machine. PH2 represents a second state. (see Fig 2).

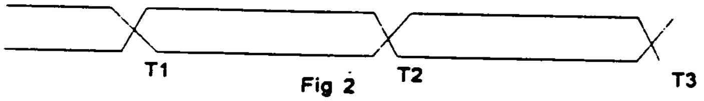
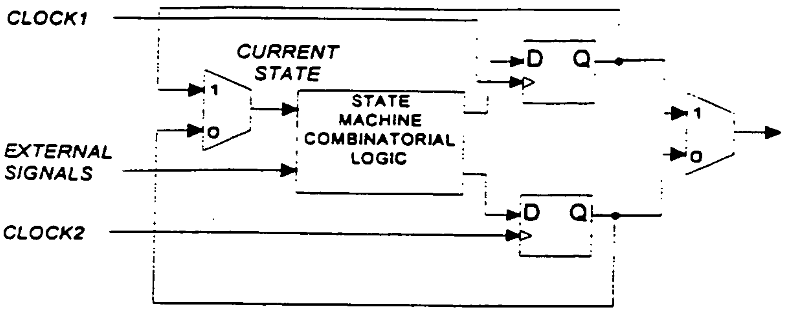


Fig 2

INVENTION DISCLOSURE

Another implementation of this concept is as follows:



This implementation has the advantage of allowing two independent clocks for the two interleaved bitstreams. This has advantages for a pipeline system where the pipe would be temporarily disabled by gating the clock. One input bitstream could be disabled while the other continues.

INVENTION DISCLOSURE

The TV systems that process both analog and digital signals require structures that can accommodate multiple input formats. The typical analog format used in off-the-air signals or cable inputs would be run through an analog-to-digital converter and then processed in a digital manner. The satellite data is digital but comes in a compressed format using either MPEG1 or MPEG2 formats. HDTV will be a digital signal in an MPEG form but using much more memory bandwidth to obtain the higher resolution picture.

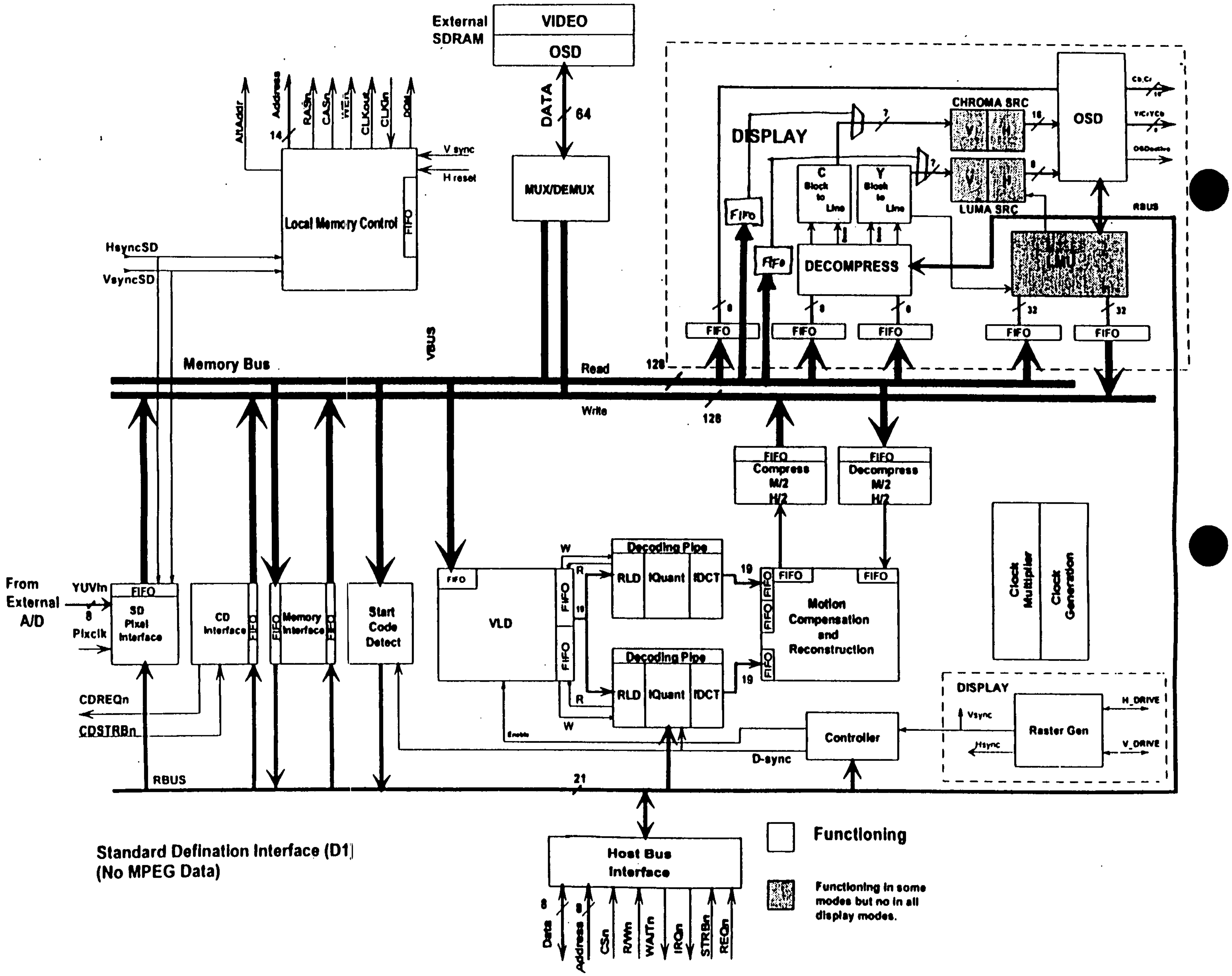
ICs that support HDTV must have an efficient method to handle the high memory bandwidth required to transfer data between the decoder and the display unit. Since this system is designed to decode MPEG with its block structure, the data sent to the display is also in a block structure. To change the data from a block structure of 8 pixels of 1line by 8 rows of lines to one line of many (1920) pixels by 8 rows, a large memory is built into the display system.

If the system is optimized for the transfer of blocks of data, we have a problem when we get data on our input that looks like our display output. The problem lies in the block-to-line conversion performed in the display section. We have two choices to follow:

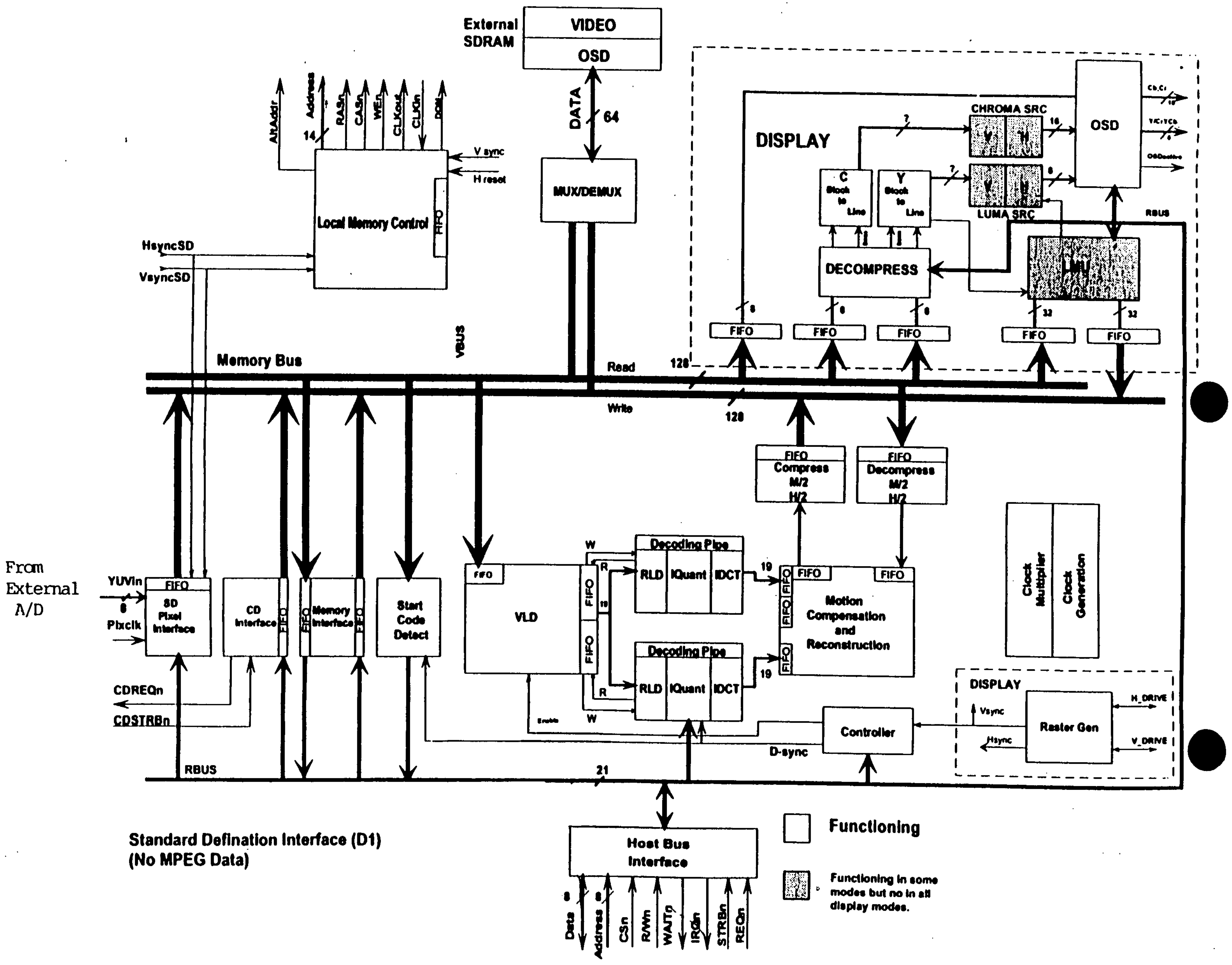
1. Bypass the block-to-line section in the display to receive data in the order that it is displayed. This is possible but it represents a rather complicated arrangement and breaks up the normal flow of data. Memory bandwidth can also enter in due to the single line loading of the memory when different display modes and scan rates may require multi-line loading of the data. The design issues become quite »mplex when designing two different techniques in the display section.

2. The second approach is to design only one display section and feed all data going to that display section in the same format. This complicates the input data handling but greatly simplifies the remaining portions of the system. If we convert the full line to the next full line of data into block form similar to the MPEG structure, the design is simplified and has very little impact on the system bandwidth.

The key concept is pre-processing of the normal D1 data into a block form doing a line-to-block technique so we can use the remaining structure of the system which contains a block-to-line converter with no further changes. This simplifies the system in design, memory bandwidth, and for debugging and test purposes.



Standard Defination Interface (D1)
(No MPEG Data)


External SDRAM
VIDEO
OSD
DATA
64
MUX/DEMUX
Local Memory Control
FIFO
AltAddr
Address
14
RASn
CASn
WEn
CLKout
CLKin
DQM
V sync
H reset
HsyncSD
VsyncSD
VBUS
Memory Bus
Read
Write
128
128
DISPLAY
FIFO
FIFO
FIFO
FIFO
FIFO
8
8
8
32
32
DECOMPRESS
C Block to Line
Y Block to Line
7
7
CHROMA SRC
V
H
LUMA SRC
V
H
16
8
OSD
Cb,Cr
16
Y/Cr/Cb
8
OSDactive
RBUS
LMU
FIFO Compress M/2 H/2
FIFO Decompress M/2 H/2
From External A/D
YUVin
8
Pixclk
FIFO
SD Pixel Interface
CD Interface
FIFO
FIFO
Memory Interface
FIFO
Start Code Detect
FIFO
VLD
FIFO
FIFO
19
W
R
R
W
Enable
Decoding Pipe
RLD
IQuant
IDCT
Decoding Pipe
RLD
IQuant
IDCT
19
19
FIFO
FIFO
FIFO
FIFO
Motion Compensation and Reconstruction
Clock Multiplier
Clock Generation
DISPLAY
Vsync
Hsync
Raster Gen
H_DRIVE
V_DRIVE
Controller
D-sync
CDREQn
CDSTRBn
RBUS
21
Standard Defination Interface (D1)
(No MPEG Data)
Host Bus Interface
Data
8
Address
8
CSn
R/Wn
WAITn
IRQn
STRBn
REQn
Functioning
Functioning in some modes but no in all display modes.

INVENTION DISCLOSURE

In a typical pipeline system, the data flows from register A to register B and then from register B to register C on each clock cycle. The data in this system is from a single source and all registers contain data from this single source. See figure 1.

In an interleaved pipeline system, the data also flows from register A to register B and then from B to C. The what is different from the typical to interleaved structure is the mixing of data from two or more sources in the datapath. At any moment in time, the data in registers A and C is from one source and the data in register B and D is from another source. The data flows together but the data between registers A and C is independent from the data between registers B and D. This concept has been used in digital processing of chroma signal for many years. Both sources can be started and stopped synchronously as long as the pipeline is started and stopped at the same time without any system problems. See figure 2.

Problem: What happens if one signal source stops sending data but the second signal source continues to send data? In systems such as HDTV, the calculations/second are critical so we can not afford to stop all of the dataflow each time one of two sources stop providing data. We must have the capability of controlling the dataflow through the pipeline for each side of the interleaved data without stopping the other side.

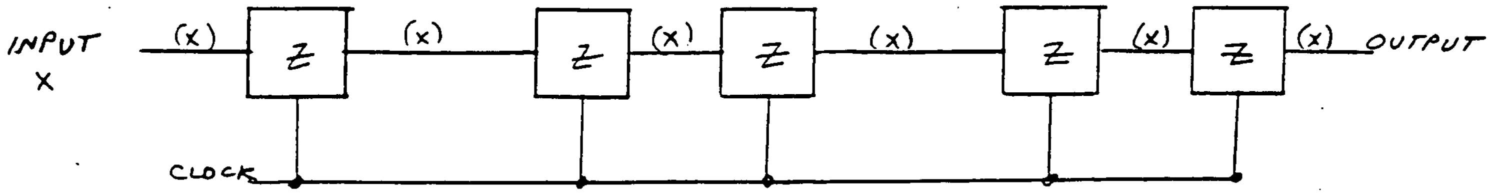
Solution: Using a mux and a control line, we are able to feedback data between every other register to keep one side of data flowing through the pipeline while the other side of data keeps repeating upon itself. Using this technique, we can interleave data and have one source completely stopped while still processing the data through the pipeline at the predicted rate. See figure 3.

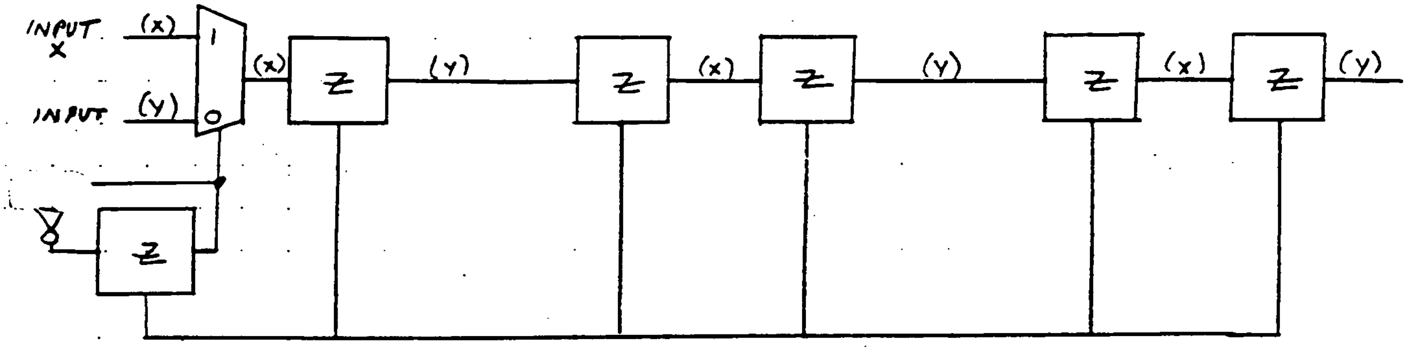
ure 4 shows one method of implementing the structure. Figure 5 shows when halt = 0 data flows through the pipeline. Figure 6 shows when halt = 1, data is circulated from output to input to keep all of the values. Note that data from input X and input Y is still maintained in the proper interleaved clock phase where every other output is from input X.

Figure 7 shows general timing when halt is pulsed. In this case, halt is only active when data from input b has stopped. All of the pipelined input b data is stopped and circulated until halt is low again and the input b data flows once again. During this time, input a data flows without any interruptions.

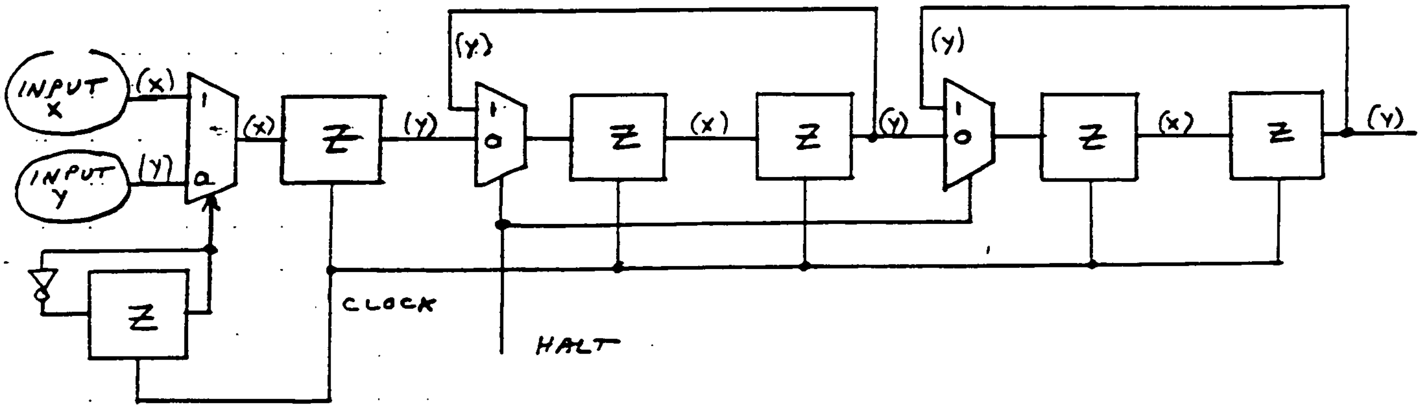
Figure 8 shows a typical application of this concept in the barrel shift circuit.

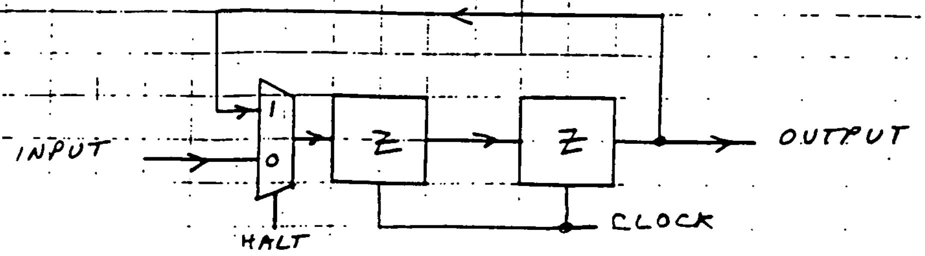
TYPICAL PIPELINE

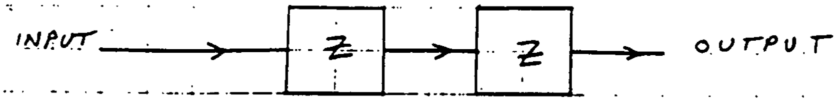




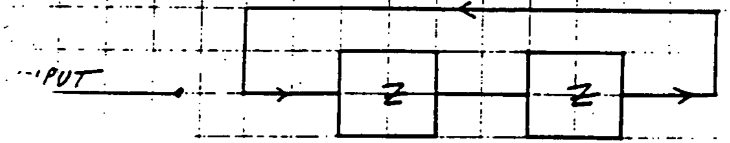
INTERLEAVED PIPELINE







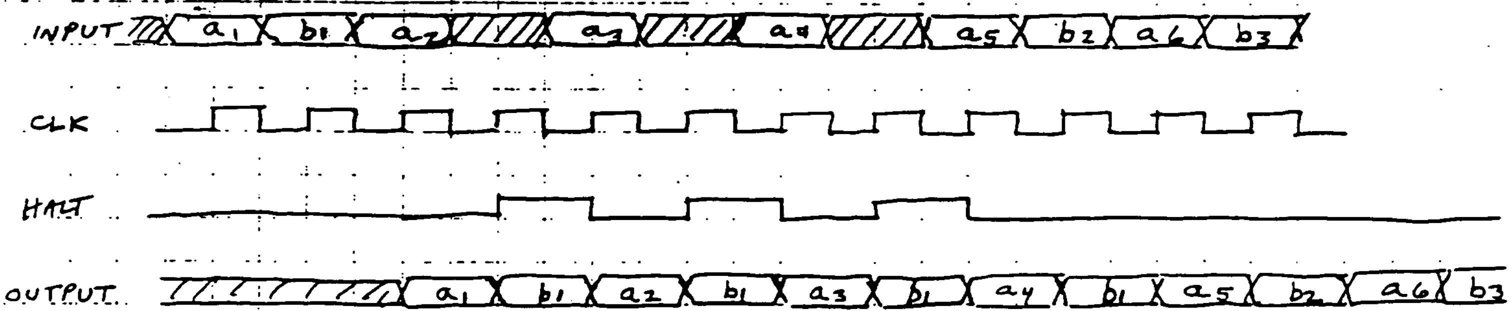
WHEN HALT = 0   OUTPUT = INPUT (z^{-2})



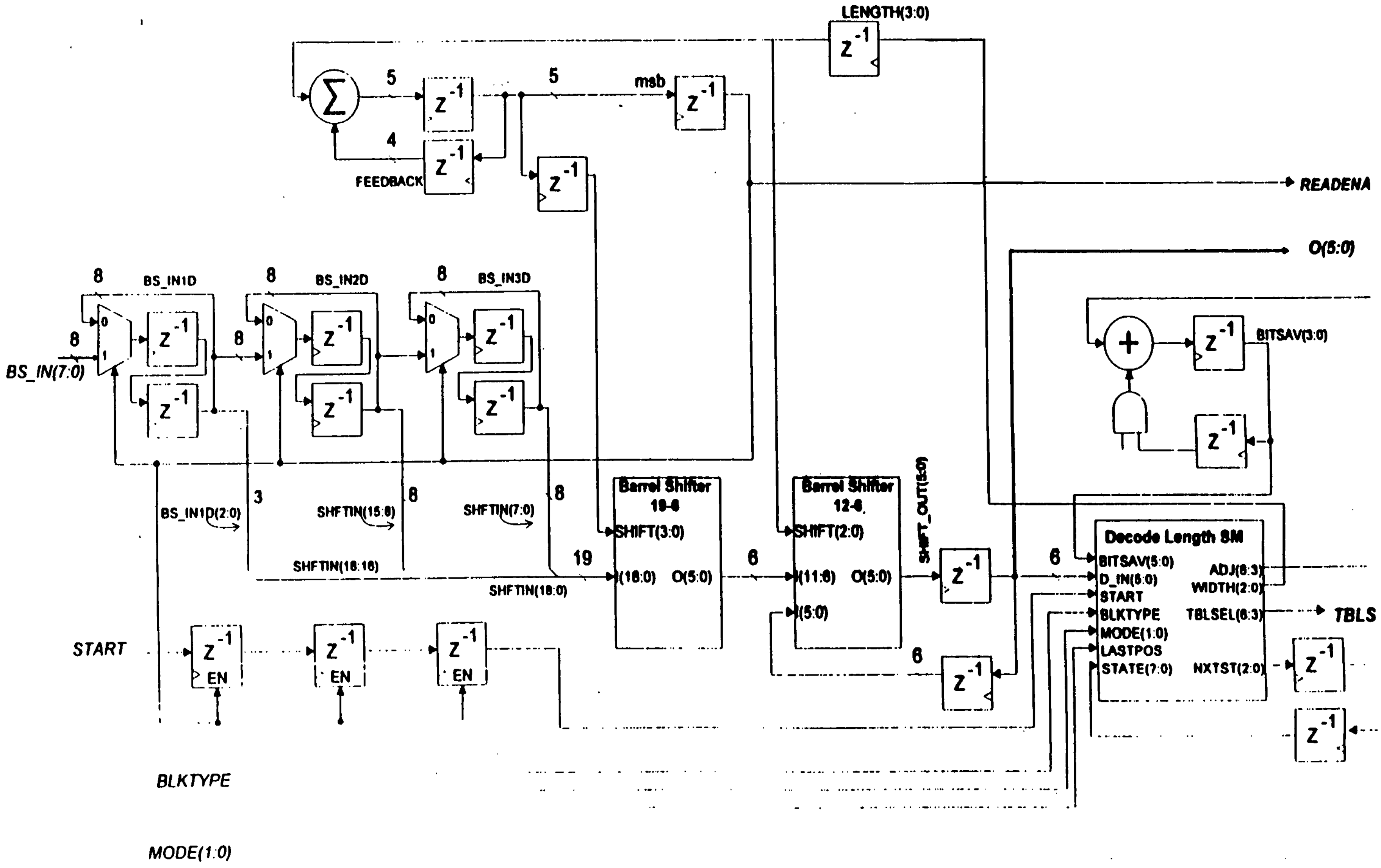
WHEN HALT = 1   OUTPUT = OUTPUT (z^{-2})

WHEN INTERLEAVED, DATA FROM A SINGLE SOURCE COMES OUT EVERY OTHER CLOCK.

IF HALT IS CHANGED EVERY CLOCK CYCLE, INPUT a_i DATA FLOWS THROUGH THE SYSTEM WHILE INPUT b_i DATA STOPS MOVING BUT STILL IS MAINTAINED.



HD MPEG IC
Memory Reduction
Barrel Shift



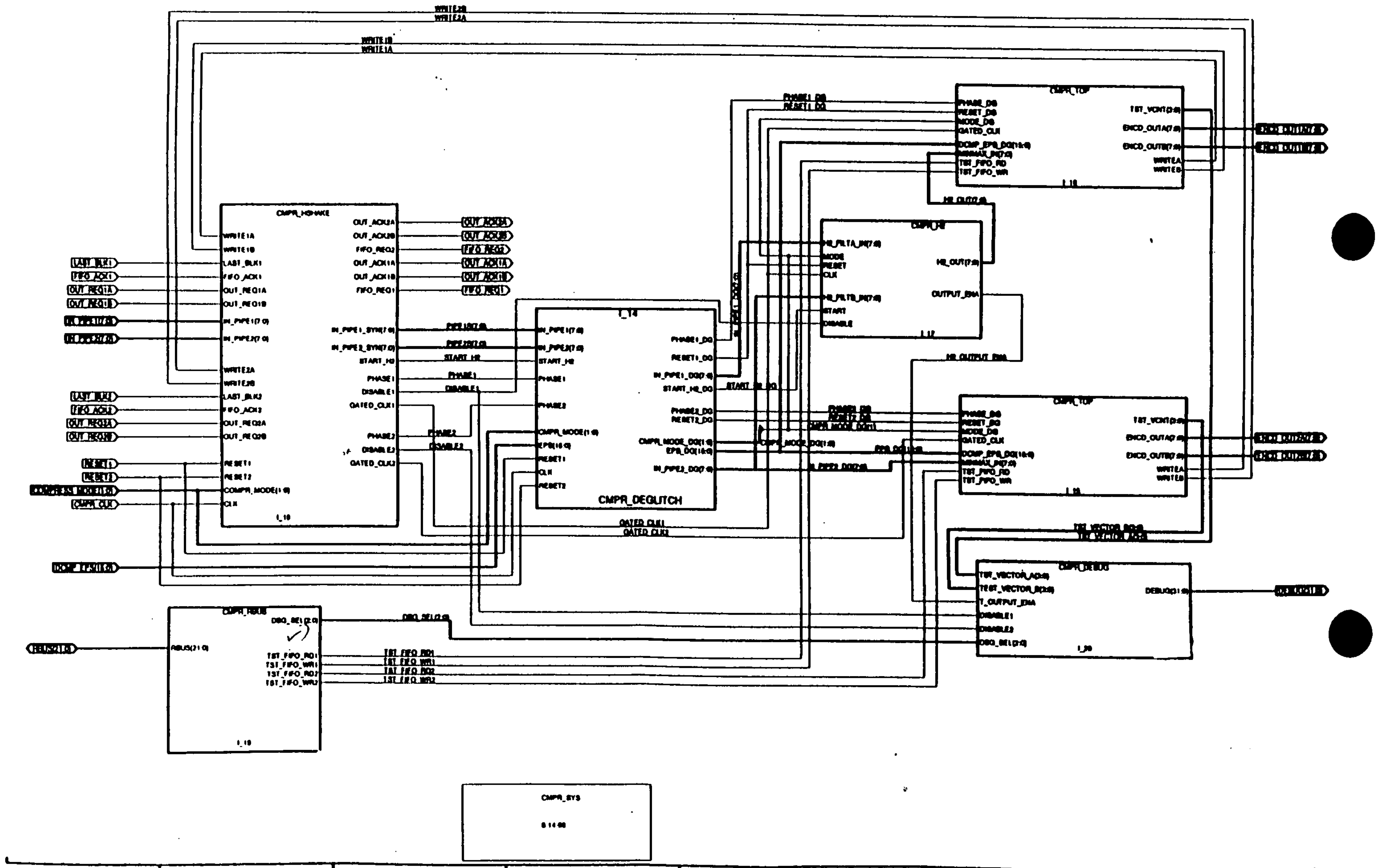
INVENTION DISCLOSURE

Figure 1 shows the normal system with full capabilities. Note that both compressors are needed and are running in the non-H/2 mode.

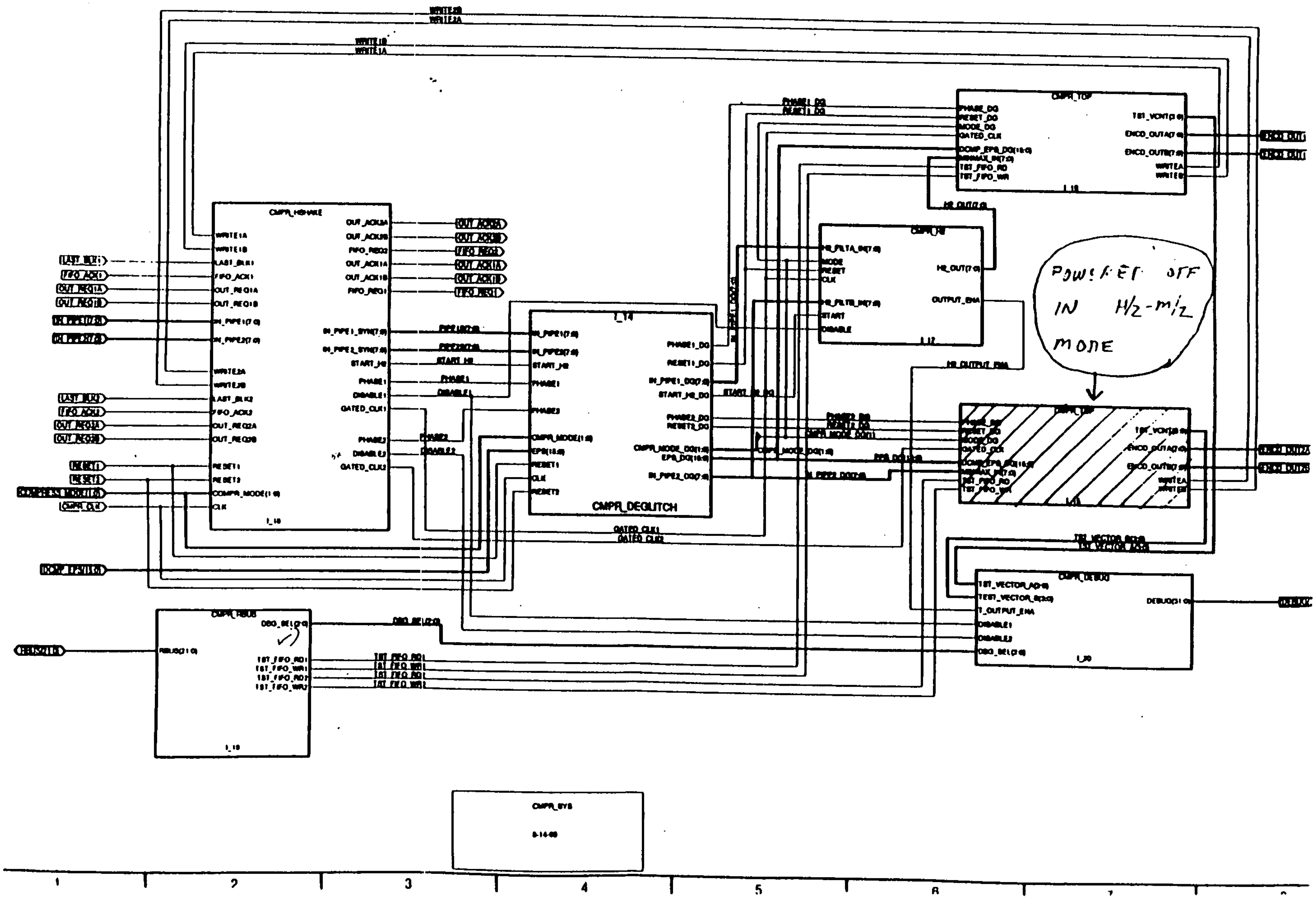
Figure 2 shows the area that can be turned off when running the H/2 mode. In a circuit that only uses H/2, the second compression block would not be needed.

The idea is simple but it is very useful for HDTV due to the power limitations of these Ics due to the frequency of the clocks, the chip area, and the large number of devices on each die.

SYSTEM
NORMAL (NON- H/2) MODE



SYSTEM
H/2 - M/2 MODE



POWERED OFF IN H/2-M/2 MODE

INVENTION DISCLOSURE

Descriptive Title: **A Block Based Adaptive Min-Max Quantizer for Memory Reduced Image Signal Processor**

(1). Background

Due to the success of standardization of digital video compression and recent advances in microelectronics and digital signal compression techniques, digital video has become an important topic in many application areas, such as consumer electronics, computer, telecommunication and medical electronics. In particular, MPEG-2 standard has received world wide acceptance and has been implemented in number of systems for broadcasting digital TV by satellite, such as DSS and DBS. At the end of last year, ATSC submitted the ATV standard to FCC for final approval, which again used MPEG2 compression techniques.

In MPEG video decoders, storage memory is generally needed to save some decoded image frames for reconstructing other frames by using motion estimation and compensation. Typically, a memory space for three frames is required for conventional decoding and the cost of the memory is always considered being a burden of the decoding system. For example, one of the ATV formats defines a frame with resolution of 1920x1080 pixels, which requires approximately 75 Mbits memory just for saving three frames. Therefore, techniques which can reduce the memory requirement for video decoding systems will be extremely useful and significant for reducing the cost, especially for consumer products, such as TV, set-top boxes, etc..

This document describes a technique which can be added to an MPEG decoder to allow MPEG decoding with only half of the normally required memory. Specifically, the invention disclosed in this document is a quantizer which will be used jointly with the invention [1] to perform block based lossy compression. Since the compression is accomplished with respect to the image pixel in the spatial domain, the complete system can be applied to all applications where frame memory reduction is demanded. The quantizer as well as the complete system has a simple structure and can be implemented by hardware with reasonable complexity. The excellent performance of the system has been achieved in many simulations including frame buffer reduction in the MPEG2 decoding loop. For 50% frame memory reduction, the picture quality is outstanding without noticeable degradation for most sequences with various MPEG encoding features.

(2). Prior Art and Overview of the Invention:

(2-a). Prior Art

The min-max quantizer disclosed in this document can be implemented with the memory reduction techniques reported in the invention [1]. In [1], a block based, multipath DPCM compression technique is described, which guarantees that the total number of bits for every compressed image block is less than a predefined boundary. Usually, the compression in [1] is characterized by the memory reduction ratio. For example, a uncompressed block of 8x8 luma data takes 512 bits in memory. If the application requires a memory reduction ratio of 50%, the compressed block will be smaller than 256 bits. In general, a new two dimensional predictor was invented and used in the system [1]. The prediction errors were quantized by two parallel paths, one was called variable-length path and the other one was called fixed-length path. The variable-length path was a lossless or near lossless compression routine but might sometime exceed the boundary of the available memory. On the other hand, the fixed-length path was a lossy compression routine with a fixed size of compressed block. These two paths were operating parallelly and concurrently. The variable-length had greater favor to be selected than the fixed-length, since it provided better reconstructed images. However, the fixed-length path was taken whenever the variable-length path failed to compress the data within the memory space boundary. The advantage of this multipath systems is that the variable path is expected to provide good picture quality in the smooth area of an image where annoying contouring artifacts are more noticeable than other areas. However, the disadvantage is that the multipath structure increases the implementation complexity which results in higher cost of hardware.

INVENTION DISCLOSURE

Descriptive Title: **A Block Based Adaptive Min-Max Quantizer for Memory Reduced Image Signal Processor**

(2-b). Overview of the Invention

In this document, a new min-max quantizer is invented for the fixed-length path in [1]. In addition, the multipath structure in [1] is simplified by removing the variable-length path, which results in a simple system shown in Fig. 1. Basically, the complete system consists of two parts: the compressor and the decompressor. The compressor takes each uncompressed image block of N bits as input, and outputs the compressed block of N_σ bits to the frame memory. On the other hand, the decompressor takes each N_σ bits of compressed data from the memory as input, and outputs N bits of reconstructed image block. Here, σ indicates the memory reduction ratio and determines the size and boundary of each compressed image block in the frame memory. The whole system is still block based which makes the image pixels in the frame memory easily accessible. This is very important for the applications like MPEG decoders where previously decoded pictures are randomly accessed for performing motion prediction. In Fig. 1, the predictor can be any type of one or dimensional predictor which meets the following condition:

$$\textit{Minimum pixel value of block} \leq \textit{Prediction} \leq \textit{Maximum pixel value of block.} \qquad (2\text{-}1)$$

The above condition is true for almost all currently used predictors in image processing.

Due to its nature, the system shown in Fig. 1 can be embedded in many image signal processors where lossy compression is permitted and memory space is concerned. For example, this system has been used in MPEG2 decoding loop for the 50% and 25% memory reduction ratios with good picture quality. It can be seen that the excellent performance of the system is mainly achieved by the min-max quantizer disclosed in this document. As being a typical DPCM compression system, the complete system is simple and can be implemented by hardware with reasonable and acceptable complexity.

Fig. 2 shows the structure of the disclosed min-max quantizer. It can be seen that the entire quantization process is based on the minimum and maximum pixel values of the current block. Although the signal coming into the quantization mapping block is the residue of the prediction, the dynamics of the input can be characterized by the minimum and maximum values of the uncompressed image block. Since the compression ratio is fixed, using the dynamics of the input to customize the quantization process can always improve the performance of the compression. This is based on the fact that the dynamics of the image signal in a block is usually a small number for the most part of the image. In this document, the term "range" is used to describe this dynamics. The minimum and maximum values along with the range of the block are obtained by the Min-Max-Range processor in Fig. 1. In this document, the range is simply obtained by taking the difference of the maximum and minimum. According to the range value, the coding controller adaptively adjust the domain of the quantizer and place all the reconstruction levels within the current domain. Here, the reconstruction levels and decision levels for every range are specially designed to be symmetrical to the mid-point of the range. Therefore, the quantizer and dequantizer only need to implement a half of the entire quantization process by the quantization map. Based on the symmetrical property, the second half of quantization mapping can be obtained from their counterparts in the first half with simple logic. This important feature of the quantizer significantly reduce the complexity and cost of the hardware. In addition, a scaling factor is used to enable two different ranges sharing one common set of quantizer's parameters. Hence, the complexity of the hardware is further reduced.

Since the minimum and maximum values of the whole block are known, negative prediction errors can be converted to positive numbers within the range before the quantization process takes place. The conversion process is accomplished by adding the range value to the negative error in the normalization block. This results in better quantization performance. Consequently, the range and minimum information are included in the compressed data for the dequantization. This is considered as the overhead of the compressed block. To compensate the overhead, some codewords have been designed with shorter size. Every time the short code is used, the overhead is reduced. In addition, a special quantization table with less number of reconstruction levels is prepared for the situation where the overhead can not be fully compensated by the shorter codewords. Tracking the overhead bits and selecting the quantization table are performed by the coding controller in Fig. 2. This controller guarantees that every block of image can be saved into memory with a predefined boundary. Since the complete system is essentially a DPCM system, the first pixel is very important to the overall performance of the prediction. Based on the range value, the first pixel processor in Fig. 2 handles the processing of first pixel. Finally, the data multiplexer writes the compressed data along with the useful range and minimum information in a predefined data format to the memory.

Figure 3 depicts the structure of the corresponding dequantizer. Compared with the quantizer, the dequantizer is relatively simple. The dequantization process starts from the demultiplexer. Basically, the demultiplexer parses the encoded parameters like the range and the minimum from each compressed block to the Min-Max-Range decoder. Based on the predefined data format, the decoder takes the corresponding bits in the compressed data to recover the minimum, the maximum and the range value. Then, these important parameters are used in the other blocks, i. e., dequantization mapping, first pixel recovery and image pixel normalization. Using a similar approach, the decoding controller for the dequantizer selects an appropriate set of dequantization parameters to implement the inverse mapping for dequantization. As being explained earlier, the negative prediction error is converted to positive number by adding the range before quantization. Therefore, the corresponding reconstructed image pixel value needs an appropriate inverse adjustment. This is accomplished in the pixel normalization process by evaluating the value of the reconstructed pixel. If the decompressed pixel is bigger than the maximum value, the reconstructed pixel will be modified by subtracting the range value. In addition, the inverse process to scale the prediction error back to its original range is done right after the dequantization mapping operation. From Fig. 3, it can be seen the entire dequantization process is also based on the minimum, the maximum and the range values.

(3) The Minimum, Maximum, and Range

From previous discussion, the minimum pixel value, the maximum pixel value and the range of the block are used in the [illegible] of the quantizer. Here, the range is defined as the difference between the maximum and the minimum pixel values. Basically, they are the key factors that control the performance of the entire quantization process in this disclosure. Generally, the range reflects the image nature and dynamics of the block. Considering real digital video data, the range of a block may be any number between 0 and 255, theoretically. However, the range is usually smaller than 256 for most of areas. Let X denotes the pixel value in a block; $X_{\min}$ and $X_{\max}$ are the minimum and maximum pixel values of the block, respectively. Then the range of the block, R, is $X_{\max} - X_{\min} + 1$. As explained earlier, the disclosed quantizer is designed in conjunction with a predictor defined in (2-1). Under the condition given by (2-1), the prediction of any pixel in the block, P, can be expressed as $X_{\min} \le P \le X_{\max}$. Therefore, the prediction error e is also bounded by the range R. It can be easily seen from the following expressions:

Since $\quad X_{\min} \le X \le X_{\max}; \qquad X_{\min} \le P \le X_{\max} \qquad$ (3-1)

then $\quad -X_{\max} + X_{\min} \le E = X - P \le X_{\max} - X_{\min};$

and $\quad -R < E < R. \qquad$ (3-2)

As a result, this block only requires a quantizer which covers any value inside the range. Therefore, the range can be considered as the domain of the prediction residue in the design of the quantizers. For the block based fixed ratio compression, using the range value to customize the quantizer can dramatically improve the performance of the quantization. This has been verified by numbers of simulations. In addition, the important impact of the range value to the performance of the quantizer can be easily shown by the fact that the quantizer has to cover any value between -255 to 255 if the dynamics of the block is not available.

In this disclosure, the range is used in customizing the quantizer for each block. Although the dynamics of the quantizer is reduced by using the range value, the range value can not be directly used by the quantizer for two reasons. First, since the decompressor needs to know the current range in order to select an appropriate dequantization mapping, the range information of each block has to be included in the compressed data. Using the exact range value may take up to 8 bits from the total target bits. Consequently, the average number of storage bits for every pixel is reduced since the size of each compressed block is fixed. Second, the hardware will become too big and complicated to implement all the possible range values. The solution to this problem is to use digitized range value. Here, total of seven different range values have been selected and implemented, which are 16, 32, 64, 128, 192,

256. Consequently, the real range value will be classified or digitized into this set. The classification is characterized by the following expression:

$$R_d = Min\{R_i \mid R_i \geq R, R_i = 16,\ 32,\ 64,\ 128,\ 192,\ 256\} \qquad (3\text{-}3)$$

The index d of the corresponding range for each block is represented by 3 bits in the compressed data.

The minimum and maximum values of the block are used here in the prediction error normalization in the quantization process, and the maximum value is needed in the pixel normalization in the process of decompression. This will be explained more in details in the following sections. The minimum value information is presented with the compressed data in each block. According to the minimum and range values, the maximum value can be easily retrieved based on the definition, which is

$$X_{\max} = X_{\min} + R - 1$$

With the same reasons for the range values, the minimum value has to be quantized and encoded into the compressed data. The quantized minimum value, $Q_{\min}$, and the quantized maximum value, $Q_{\max}$, along with the range has to meet the following conditions.

$$Q_{\min} \leq X_{\min} \quad \text{and} \quad Q_{\max} \geq X_{\max}; \qquad (3\text{-}4)$$

where $Q_{\max} = Q_{\min} + R_d - 1$.

These conditions are very important in the data normalization which is explained in the next section. Here, the minimum value is quantized into 8 predefined numbers for the digitized ranges of 32, 64, 96, 128, 192. Every one of these ranges has a different set of quantized minimum values with common number 0, which results from the above constrains. The index of the quantized minimum is also represented by 3 bits in the compressed data. Therefore, the quantized minimum value can be retrieved from the compressed data by first using the range information to get the corresponding set, then searching for the correct value inside the set with the decompressed index. The range of 16 and 256 are different. For the range of 16, the real minimum value is used in the compressed data which takes 8 bits. Since the dynamics of this range is relatively small, it is hard to make the quantization with the constraints in (3-4). This can be seen from the following explanation. For the range of 256, the minimum is always 0. Therefore, no extra bits are needed for encoding the minimum information in the compressed data. The minimum of 0 is always used in the decompression if the range value is 256.

According to the discussion above, the quantized minimum is related to the digitized range R_d and is chosen based on the following expression:

$$Q_{\min} = \underset{i}{Max}\{Q_{\min}(R_d,\ i) \mid Q_{\min}(R_d,\ i) \leq X_{\min};\ 0 \leq i \leq 7\} \qquad (3\text{-}5)$$

where $Q_{\min}(R_d,\ i)$ is the i-th number on the list of the preselected minimums for the range of R_d. The $Q_{\min}(R_d,\ i)$ can be simply derived as:

$$Q_{\min}(R_d, i) = [i \cdot \tfrac{256-R_d}{7}]; \qquad 0 \leq i \leq 7 \qquad (3\text{-}6)$$

where $[x]$ takes the integer part of the number x .The above calculation results in a set of digitized minimums with a constant step size for every range. This could be modified depending on the application requirements.

Sometimes, the quantized minimum $Q_{\min}$ in (3-5) may not results in a proper $Q_{\max}$ which meets the second condition in (3-4). In this situation, the upper level of the digitized range R_d will be used. For example: $X_{\min}$ is 100, $X_{\max}$ is 140. The difference is 40. Based on (3-3), the digitized range R_d is 64. If the quantized minimum $Q_{\min}$ is 81, the $Q_{\max} = Q_{\min} + R_d = 81 + 64 = 144$ is bigger than 140, which meets the second condition in (3-4). If the $X_{\max}$ is 150 instead, the R_d is still 64. However, the $Q_{\max}$ of 144 will be too small and fail the second part of (3-4). Consequently, the digitized range R_d of 96 will be taken and the $X_{\min}$ will be quantized by a new set of minimums associated with the R_d of 96. Figure 4 summarizes and shows a typical implementation.

The detailed bits usuage for encoding the digitized range and the minimum values is given in Section (6).

(4). Normalization of Prediction Errors and Reconstructed Pixels

According to (3-1) and (3-2), every pixel value, predicted pixel value and reconstructed pixel value for each block should be inside the range between the quantized minimum and the quantized maximum. Therefore, the prediction error can be any number in the domain of (- R_d , R_d) (3-3). Since both boundary pixel values are known in both compression and decompression process for every block, a negative prediction error can be converted to a positive number by adding the current range value R_d before the quantization. Then, the normalized prediction error is quantized and encoded in the compressed data. In the decompression process, the dequantized prediction error is added to the corresponding prediction value from the predictor to get the reconstructed pixel. For the converted prediction error, the reconstructed pixel value will be bigger than the quantized maximum. Therefore, it can be easily identified in decompression. Then, the corresponding reconstructed pixel will be converted back to the correct value by subtracting the R_d . The advantage of the normalization process can be easily seen from the fact that all the normalized prediction errors are positive numbers in the domain of [0, R_d) other than (- R_d , R_d). As a result, the quantization resolution is enhanced.

Let X be any pixel in the block with $Q_{\min}$, $Q_{\max}$ and R_d . P is the prediction of the X . According to previous discussion,

$$Q\min \le X,\ P \le Q\max\,;$$
$$-R_d < E < R_d$$

E denotes the prediction error.

In the following discussion, X_r denotes the reconstructed pixel; N_Q denotes the quantization noise; $Q[x]$ denotes the quantized value of x .

For any positive E ,

$$X_r = Q[E] + P = E + N_Q + P = X - P + N_Q + P = X + N_Q \qquad (4\text{-}1)$$

For any negative, E

$$X_r = Q[E + R_d] + P = E + R_d + N_Q + P = X - P + R_d + N_Q + P = X + R_d + N_Q \qquad (4\text{-}2)$$

If $N_Q = 0$, it can be seen that X_r in (4-1) is the original value of X which is inside the block boundaries, while $X_r = X + R_d$ in (4-2) is always bigger than the upper bound of the block, $Q_{\max} = Q_{\min} + R_d - 1$. It can be easily proven that the reconstructed pixel will be greater than the $Q_{\max}$ if and only if the negative prediction error has been converted. Therefore, the converted negative prediction error can always be identified by comparing the reconstructed value with the $Q_{\max}$. However, if N_Q is not zero, the X_r in (4-1) may be bigger than $Q_{\max}$ due to large positive quantization noise, while the X_r in (4-2) may be smaller than $Q_{\max}$ due to a negative quantization noise with large amplitude. For decompressing data correctly, the compressor has to guarantee that the normalized prediction error will be quantized to a number which will not be misinterpreted by the decompressor. From (4-1) and (4-2), it can be seen that adjusting the quantization output can modify the quantization error. This is explained in details as following.

Assume a quantizer has M reconstruction levels. Let Q_i, $0 \le i \le M-1$, denotes the reconstruction level, which can be ordered as

$$Q_0 < Q_1 < Q_2 \ldots\ldots < Q_{M-1}$$

Assume Q_i is the quantized value of E, $E>0$, and the quantization noise $N_Q = Q_i - E$. The misinterpretation of Q_i being an quantized negative prediction error with conversion will happen if and only if when $N_Q > 0$ and $X + N_Q > Q_{\max}$. According to the nature of the quantization reconstruction levels, the Q_{i-1} will always be smaller than E and the corresponding N_Q will be smaller than 0. Therefore, selecting the lower level next to the current reconstruction level whenever the reconstructed quantization level is bigger than the $Q_{\max}$ can guarantee that the positive prediction error can be interpreted correctly.

For a negative E, Q_i represents the reconstruction value of $E + R_d$. Based on (4-2), Q_i will be misinterpreted as a quantized positive prediction error with no conversion if and only if $N_Q < 0$ and $X + N_Q + R_d \leq Q_{\max}$. Similarly, this can be avoided by taking the Q_{i+1} as the output since Q_{i+1} is always bigger than E and N_Q is always bigger than 0. It is worth to note that the quantization performance will be degraded by selecting the neighboring reconstruction levels. Although the modifications are necessary when the problems exist, they are not needed in the most of time during the course of real video compression. Generally, there are two different ways to implement the above verification in the process of quantization. First, design a quantization mapping circuit which provides both Q_i and the next level either Q_{i+1} or Q_{i-1} depending on the sign of the prediction error. This mapping block is followed by a selection block which compares $Q_i + R_d$ with the maximum value $Q_{\max}$. If Q_i meets the conditions by which the decompressor can correctly reconstruct the pixel, then the selection block will take Q_i as the output of the quantization. Otherwise, the next value from the quantization mapping will be selected. It is obvious that the complexity of the quantization mapping in this proposal is more than twice the one for a conventional quantization with Q_i as the only output. As the second proposal, a simple method has been developed as shown in Fig. 5. In Fig. 5, the MAP1 is a simple conventional quantization with single output

The MAP2 is a complementary map which can be described as:

$$Y_i = MAP2(Q_i) = Q_{i-1} \cdot sign(E) + Q_{i+1} \cdot \overline{sign(E)}; \qquad 1 \leq i \leq M-2 \qquad (4\text{-}3)$$

with
$$Y_0 = MAP2(Q_0) = Q_1 \cdot \overline{sign(E)}$$

and
$$Y_{M-1} = MAP2(Q_{M-1}) = Q_{M-2} \cdot sign(E)$$

where $sign(x)$ is 1 if x is a positive number and 0 otherwise.

Based on (4-3), the MAP2 is much simpler than MAP1 in terms of the complexity of the hardware. The selection part in Fig. 5 will take the current quantized value if the quantization error passes the evaluation. Otherwise, the output of the complementary map Y_i will be taken instead.

As mentioned before, the quantization maps are shared by the normalized signals from different ranges. This is implemented by scaling the signals from different ranges into a common range before the quantization takes place in the compression process. On the decompression process, the corresponding scaled data will be scaled back to its original range with the same scaling factor. Specifically, a scaling factor of two is used in this disclosure. For example, the range of 32 is sharing the quantization maps with the range of 64 by multiplying any normalized prediction error in the range of 32 by 2 before the quantization. Then, the quantized data will be divided by 2 on the dequantization process, provided the range value is 32. This will obviously reduce the complexity and cost of the hardware for the quantization/dequantization maps which usually are the dominant part of the entire circuits. Beside the above ranges, any number in the range of 96 is scaled up by a factor 2 to share the quantization maps with the range of 192. Also, the range of 128 and the range of 256 are sharing a common set of maps.

(5) Mid-Point Symmetric Quantization Tables

Since the negative prediction errors are converted to positive numbers before being quantized, the quantizations involved in the compression process are always performed in the domain like $[0,\ R_d)$. However, the actual prediction error can be any integer number between $(-R_d,\ R_d)$. It is well known that the distribution of the prediction error is symmetrical around 0 and decreases fast along two directions. Therefore, the quantized prediction errors are required to be symmetrical around zero too. For instance, if b ($b > 0$) is quantized to c ($c > 0$), then $-b$ is required to be quantized to $-c$. According to the previous discussion, the negative number $-b$ will be normalized to $R_d - b$ before quantization takes place. Therefore, $R_d - c$ should be the quantized value for $R_d - b$ since R_d will be subtracted from $R_d - c$ at the pixel normalization block in the decompression. Also, if a ($a > 0$) is quantized to 0, then $-a$ should also be quantized to 0. This indicates that 0 should be one of the reconstruction levels for every quantizer. However, the negative number $-a$ is normalized to $R_d - a$, and R_d can not be the reconstruction level since it is out of the range. In this situation, $-a$ will be quantized to 0 directly. As results, if prediction error is negative, number $R_d - a$ should be quantized to 0. Based on above discussions, $R_d / 2$ needs to be one of the quantization reconstruction levels for maintaining the symmetrical property.

For a quantizer with M reconstruction levels, let Q_i denotes the *i*-th reconstruction level; D_i denotes the *i*-th decision point; its domain is [0, R_d). The Q_i and D_i can be expressed as:

$$0 = Q_0 < Q_1 < Q_2 \ldots\ldots < Q_{M-1} \le R_d - 1;$$
$$0 \le D_0 < D_1 < \ldots\ldots < D_{M-1} \le R_d - 1;$$

The quantization can be described as following:

. For any number in the [0, D_0], the reconstruction level is $Q_0 = 0$;

For any number in the (D_{i-1}, D_i], the reconstruction level is Q_i, $1 < i < M - 1$;

(c). For any positive prediction error, any number in the domain of (D_{M-1}, $R_d - 1$] will be quantized to Q_{M-1} ;

(d). For any negative prediction error, converted numbers in the domain (D_{M-1}, $R_d - 1$] will be quantized to 0.

For a quantizer with those previously discussed symmetrical characteristics, following relations have to be maintained in making the reconstruction and decision levels:

(a). M has to be an even number;

(b). $D_i + D_{M-1-i} = R_d - 1; \quad 0 \le i \le \frac{M}{2} - 1$;

(c). $Q_0 = 0; \quad Q_{\frac{M}{2}} = \frac{R_d}{2}$;

(d). $Q_i + Q_{M-i} = R_d; \quad 1 \le i \le \frac{M}{2}$;

From above relations, it can been seen that the reconstruction level Q_{M-i} can be obtained by subtracting Q_i from R_d , for $1 \le i \le \frac{M}{2}$. Therefore, the dequantization mapping block only needs to implement dequantizations of $\frac{M}{2} + 1$ levels for every range value. The rest levels can simply be derived from these dequantized levels. On the other hand, the quantizer is also able to reduce the hardware implementation for the quantization mapping block by using the symmetrical property given in (b). This is a very important nature of the symmetrical quantizer presented in this disclosure. It can simplify the hardware implementation.

In addition to all the aspects previously discussed, optimal quantizers are designed for the domain of [0, $\frac{R_d}{2}$] in this invention in the sense of minimum mean square error of quantization.

(6). Encoding/Decoding the First Pixel

For DPCM type of image compression systems, the precision of the first pixel in the compressed data is very important to the performance of the predictions for the surrounding pixels. In a smooth area, the noise with the first encoded pixel may cause significant annoying artifacts. However, spending more bits on the first pixel than the average bits for the rest of the pixels will also increase the overhead bits. Therefore, a compromise is considered between the precision and the number of bits for the encoded first pixel. In this invention, the quantized minimum value $Q_{\min}$ is used as the prediction of the first pixel. The encoding approach is described as following.

Let X_0 denote the original first pixel value, QX_0 denote the encoded first pixel, and RX_0 denote the recovered first pixel. Then, QX_0 is obtained as

$$QX_0 = [(X_0 - Q_{\min}) / 2] \qquad (6\text{-}1)$$

and RX_0 is reconstructed as

$$RX_0 = (QX_0 + Q_{\min}) \cdot 2 \qquad (6\text{-}2)$$

where $[x]$ denotes the integer part of x.

According to (6-1) and (6-2), the maximum difference between X_0 and RX_0 is 1, which meets the precision requirement. Sin $X_0 - Q_{\min}$ is within the range R_d, QX_0 is always smaller than a half of the R_d. This results in saving bits for encoding the first pixel for the small range values. The number of bits for encoding QX_0 for every range value is listed in the following table.

Table 1: Bits for encoding the first pixel

Range Value R_d	6	32	64	96	28	92	56
Bits for QX_0	3	4	5	6	6	7	7

It is worth to mention that the operations described in (6-1) and (6-2) can be easily implemented by bit shifting. In addition, the operation guarantees that RX_0 is still inside the range R_d with the same minimum $Q_{\min}$, which is very important for the entire quantization process.

(7). Encoding/Decoding Controller for Bit Allocation and Overhead Compensation

In this invention, a luma block contains 8x8 pixels; and a chroma block has 4x4 pixels. Each luma or chroma pixel is represented originally by 8 bits. Therefore, a luma block takes total of 512 bits of memory in storage without compression, and a chroma block takes 128 bits. In this invention, compression ratio is defined as the percentage of the compressed block size over the original block size. Hence, the target number of bits, N_σ, of a compressed block can be calculated by multiplying the compression ratio σ with the original block size. Knowing the target number of bits of a compressed block, averaging bits for each compressed pixel can be obtained by dividing the N_σ by the total number of pixels in the block. This average number is very important since it indicates the average length of the codeword for every pixel. For a quantizer with fixed length codewords, the length of the codewords gen ly determines the total number of reconstruction levels available.

Since the range and minimum information needs to be present in the compressed data for decompression, a few number of bits [illegible] to be taken out from the target number of the compressed block. In addition, the first pixel of every block is encoded with more bits than the average as explained early. Therefore, the overhead, N_o, is defined as the total number of bits for encoding the range, the minimum and extra bits for the first pixel. To guarantee the compressed block does not exceed the target number, overhead bits has to be compensated by the encoding process in every block. In this invention, the following approach is used.

INVENTION DISCLOSURE

Descriptive Title: **A Block Based Adaptive Min-Max Quantizer for Memory Reduced Image Signal Processor**

The optimal quantization tables designed in this invention have fixed length codewords with two exceptions. Although most codewords are with the average code size derived from the compression ratio, these two special codewords are 1 bit shorter than the rest of the codewords and are assigned to two most possibly used reconstruction levels. Therefore, each time these two codeword are used, one bit is saved and the number of overhead bits is reduced by one. However, this does not guarantee that for every block all the overhead bits can be saved. Therefore, "low-priority" is defined to the pixels at N_o different locations for each block with N_o overhead bits. They are usually located at the end of the encoding process. Basically, another special quantization table is used to quantize the low-priority pixels if it is needed. All the codewords of this special quantization table are 1 bit shorter than the average code size. To monitor the progress of overhead compensation and select an appropriate quantization table, a controller is designed in both Min-Max Quantizer and Min-Max Dequantizer. The controller is used to track the current number of overhead bits and the number of saving for every pixel in the block. According to the controller, the low resolution quantization table will be activated for quantizing the low-priority pixels if and only if the savings are not enough and the possible overflow may occur. On the other hand, if the image signal represents a smooth area, the quantization table with higher quantization resolution is expected to reduce the quantization artifacts. According to the nature of the predictor, the prediction errors are always small numbers for smooth areas. Therefore, the two special short code words should be assigned to two reconstruction levels for small values to improve the quantization quality. Consequently, more savings are obtained in the smooth areas and the low-priority pixels could be quantized with the normal quantization table. In addition, another quantization table with codewords 1 bit longer than the average code size is also assigned for the situations where savings are more than the overhead bits. This results in a better overall quantization performance. However, the range of 16 does not need this high resolution table since any normalized prediction in this particular range can be exactly represented by 4 bits without any quantization noise. Therefore, the 4 bits normalized quantization error can be used directly in the compressed data once it is possible.

The approach presented here has been implemented for the application requiring a 50% compression ratio. In the 50% mode, the average code size is 4. For the range value of 64, 192 and 256, a set of quantization tables is designed. They are the regular, the low-resolution and the high-resolution table. Because of the two short codewords, a quantization table with total of 14 reconstruction levels is designed as the regular table for the normal pixels. Twelve levels are with 4 bits codewords, and the other two with 3 bits. For the possible low-priority pixels, a quantization table with 8 reconstruction levels is designed as the low-resolution table whose codewords are 3 bits long. Another quantization table with better resolution is also designed as the high-resolution table which has 26 reconstruction levels. Twenty four levels are represented by 5 bits codewords and the other two use 3 bits codewords. For the range of 16, only the regular and the low-resolution tables are designed which have the same definition as the others. A sample controller is given in the Fig. 6a, 6b, 6c. Three registers are used in this sample controller. They are the *bitsaving*, *overhead* and *pixel_counter*. They are initialized as:

bitsavng=0; *overhead=* N_o *;* *pixel_counter=0;*

To summarize this document, Figure 7 shows the format of the compressed data, and Table 2 provides the detailed bits spending on encoding the range, the minimum and the first pixel for each digitized range value discussed here.

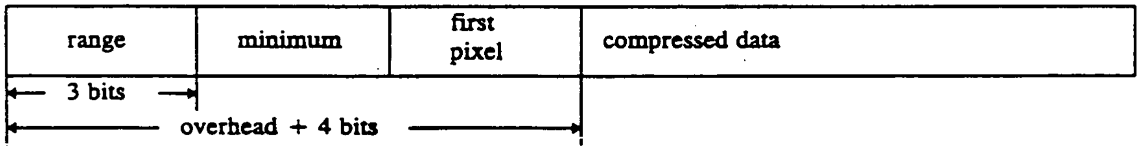


Fig. 7. The compressed data format

Table 2. Summary of Bits Spending on the Range, Minimum and First Pixel

Range Value	16	32	64	96	128	192	256
Range Bits	3	3	3	3	3	3	3
Minimum Bits	8	3	3	3	3	3	0
First Pixel Bits	3	4	5	6	6	7	7
Overhead Bits	10	6	7	8	8	9	6

(7). Simulations

The disclosed quantizer has been simulated in the HDTV memory reduction application. Intensive simulations have been done with many CCIR601 and HDTV sequences with target compression ratio σ =50%. The whole system provides excellent picture quality with minor random noise which can be observed in high detail areas in some sequence. Overall, the picture is certainly acceptable.

(8). Conclusion

This document describes a block based adaptive min-max quantizer with mid-point symmetric tables. This quantizer can be appiied in the image processing applications where storage memory reduction is desired.

(9). Reference

[1]. "Memory Management for an Image Signal Processor", Disclosed as RCA 87,791.

INVENTION DISCLOSURE

Descriptive Title: **A Block Based Adaptive Min-Max Quantizer for Memory Reduced Image Signal Processor**

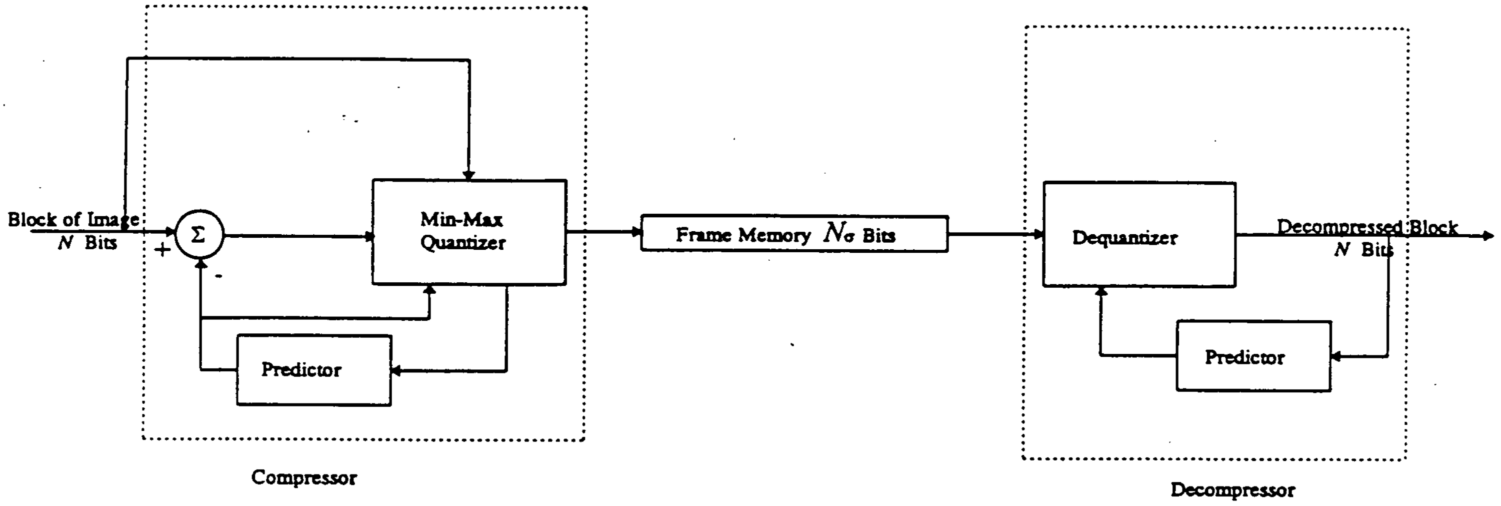


Fig. 1. Memory Reduction System

INVENTION DISCLOSURE

Descriptive Title: **A Block Based Adaptive Min-Max Quantizer for Memory Reduced Image Signal Processor**

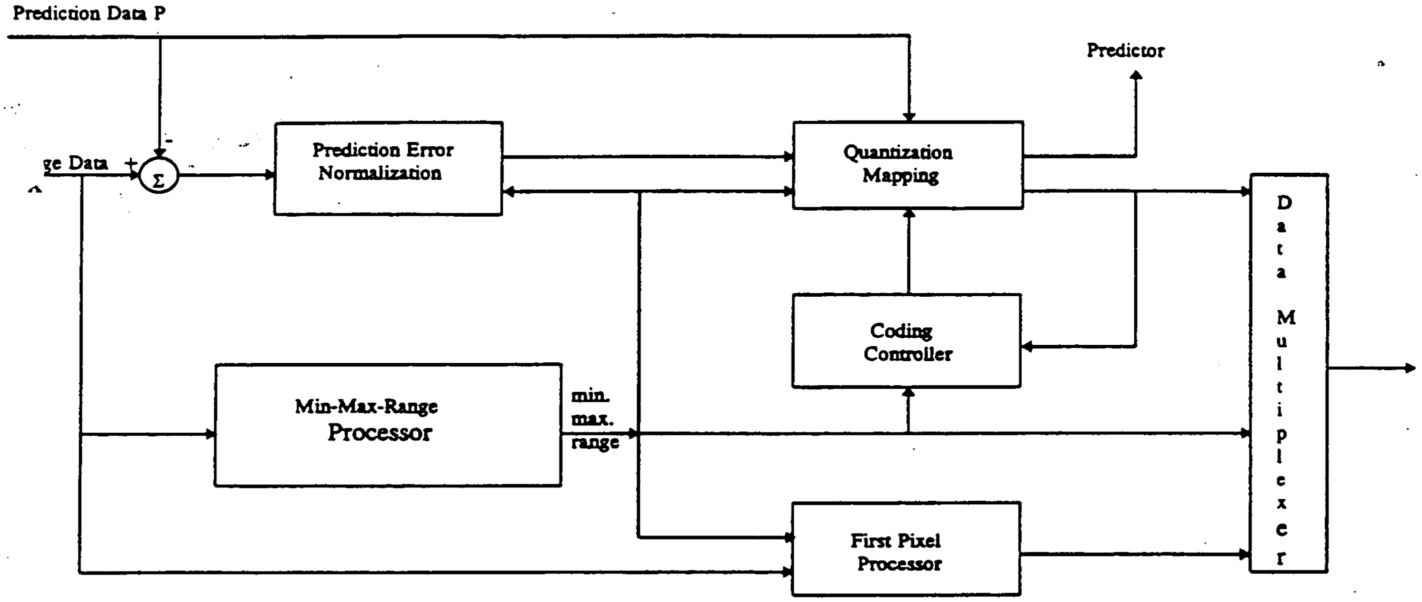


Fig. 2. Min-Max Quantizer

INVENTION DISCLOSURE

Descriptive Title: **A Block Based Adaptive Min-Max Quantizer for Memory Reduced Image Signal Processor**

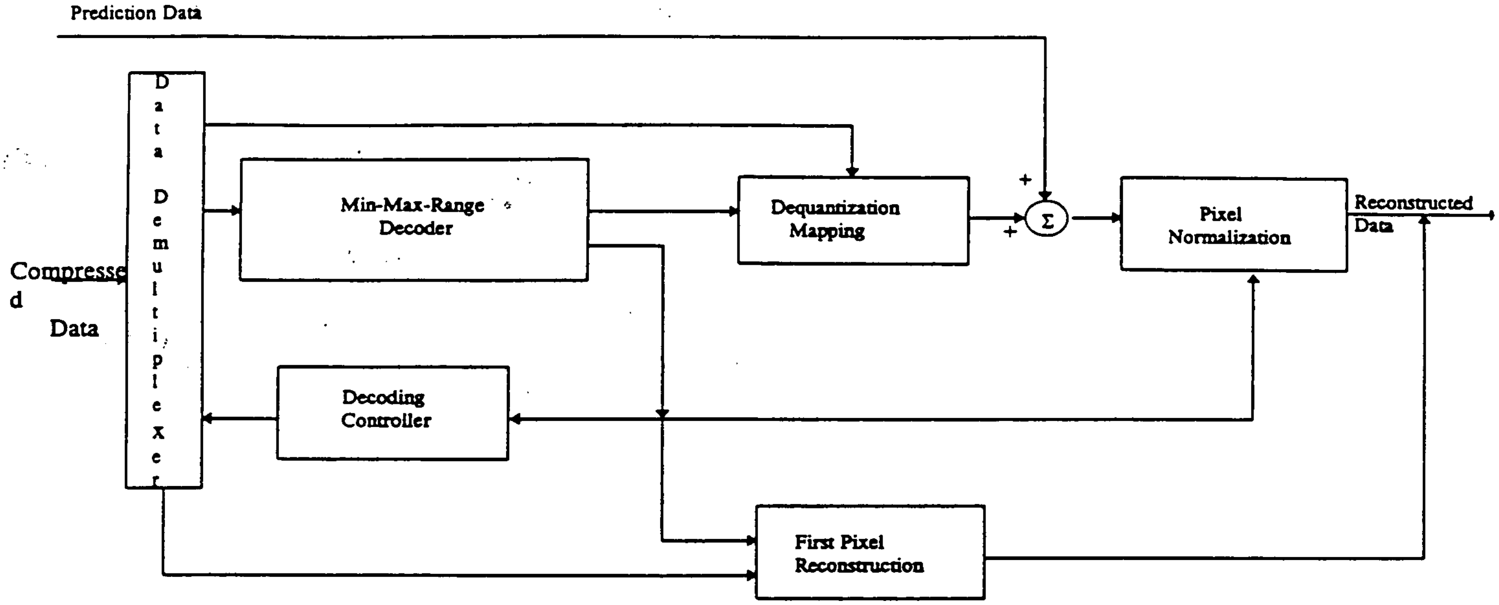
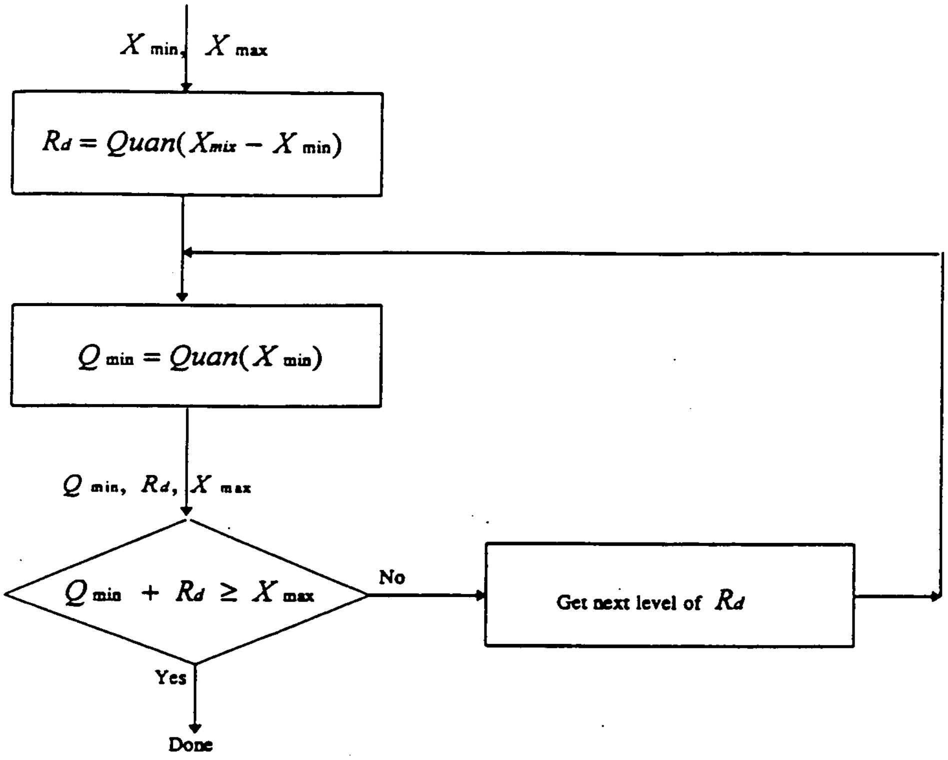


Fig. 3. Min-Max Dequantizer

INVENTION DISCLOSURE

Descriptive Title: **A Block Based Adaptive Min-Max Quantizer for Memory Reduced Image Signal Processor**


$X_{\min}$, $X_{\max}$
$R_d = Quan(X_{mix} - X_{\min})$
$Q_{\min} = Quan(X_{\min})$
$Q_{\min}$, R_d, $X_{\max}$
$Q_{\min} + R_d \geq X_{\max}$
No
Get next level of R_d
Yes
Done

Note: $Quan(\ )$ indicates the process of quantization.

Fig. 4. Min-Max-Range Processor

INVENTION DISCLOSURE

Descriptive Title: **A Block Based Adaptive Min-Max Quantizer for Memory Reduced Image Signal Processor**

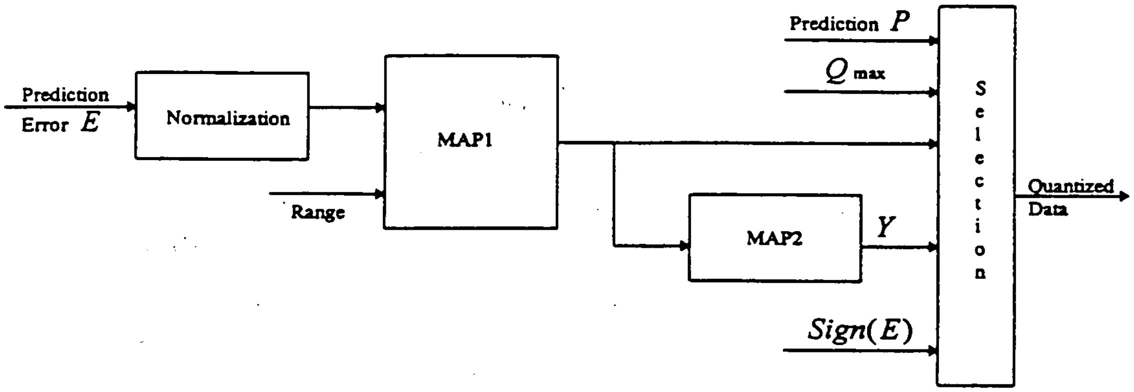
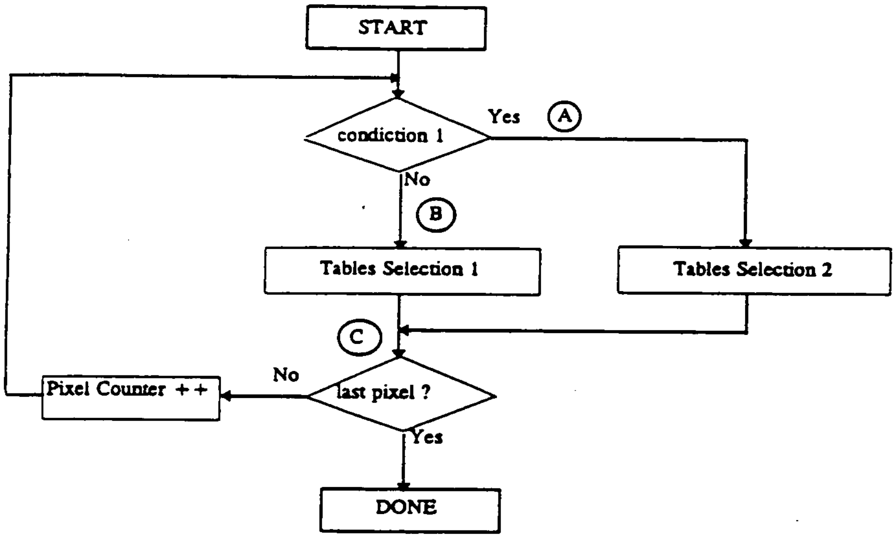


Fig. 5. Quantization Mapping

INVENTIÒN DISCLOSURE

Descriptive Title: **A Block Based Adaptive Min-Max Quantizer for Memory Reduced Image Signal Processor**



Condition 1: $bitsaving > overhead$ or $bitsaving \geq overhead$ for the range 16.

Fig. 6a. Encoding/decoding quantization controller

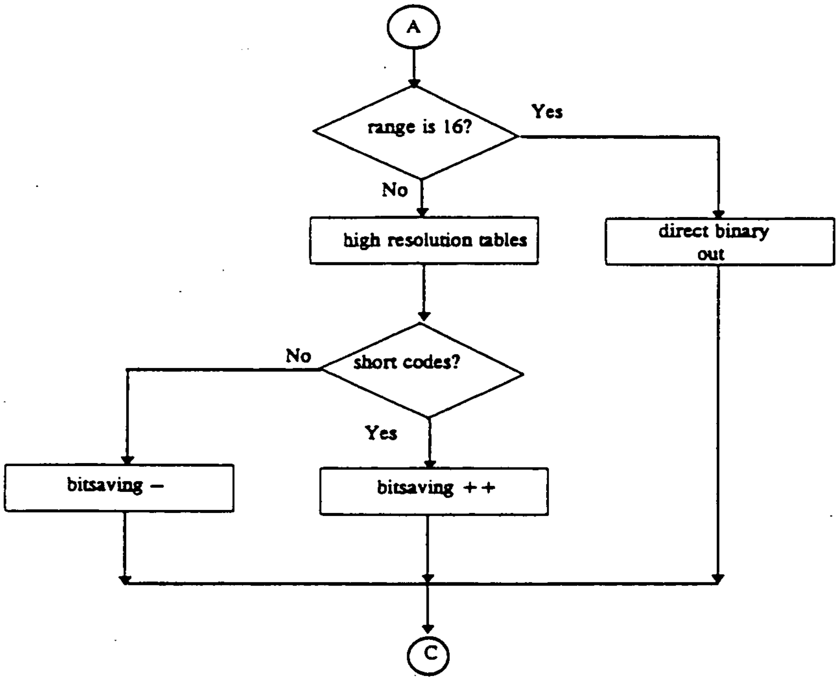


Fig. 6b. Table selection 2 in the quantization controller shown in Fig. 6a.

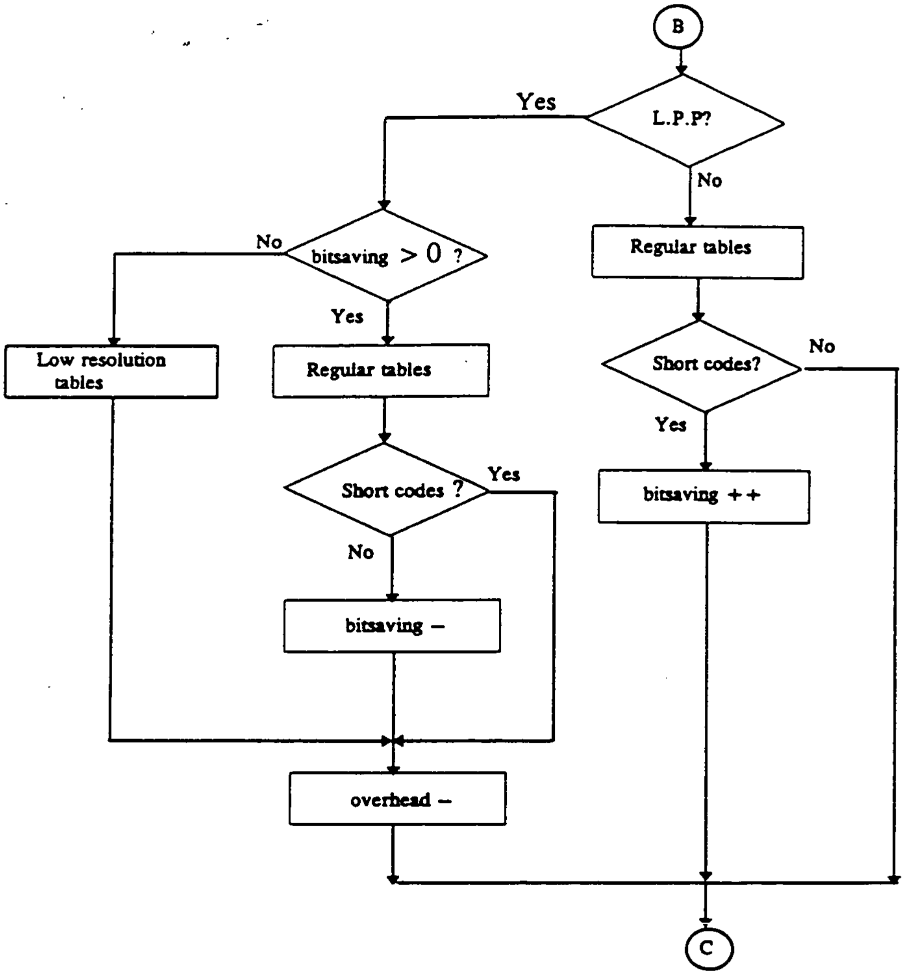


Fig. 6c. Table selection 1 in the quantization controller shown in Fig. 6a.

A quntizer with mid-point symmetric quantization tables was described in Section (5). Here, the coding scheme used in the compressor and decompressor is described with the hardware implementation diagram. In addition, an example table for the range of 64 is provided to illustrate how the hardware can be simplified by the design.

(1). Notations

i: quantizatoin level index;
Di: the ith decision point;
Qi: the ith reconstruction level;
Ci: the codeword for the ith reconstruction level;
M: total number of levels;
Rd: the quantization range;

(2). Summary of the quantization Rules to make the tables with the mid-point symmetrical property

(2-1). M has to be an even number;

(2-2). $D_i + D_{M-1-i} = R_d - 1; \quad 0 \le i \le \frac{M}{2} - 1;$

(2-3)). $Q_0 = 0; \;\; Q_{\frac{M}{2}} = \frac{R_d}{2};$

(2-4). $Q_i + Q_{M-i} = R_d; \quad 1 \le i \le \frac{M}{2};$

(3). *Encoding Scheme*

(3-1). $C_1 = 0$; $C_{M-1} = 1$; (short code words).

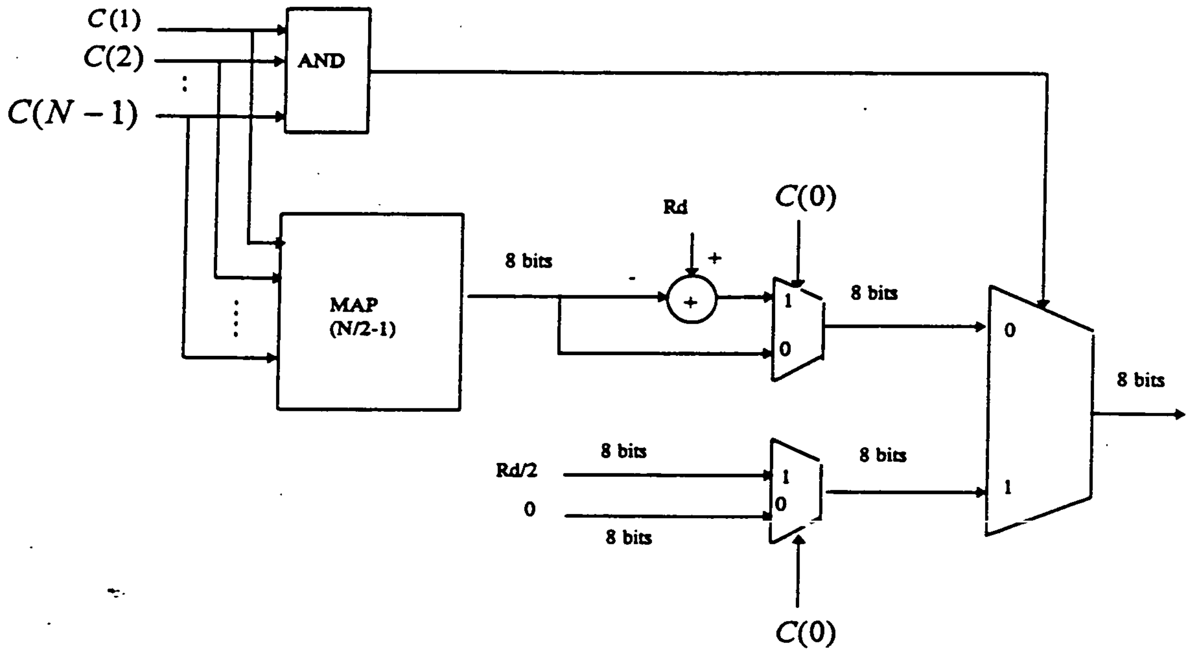
(3-2). $C_0 = M - 2$; $C_{\frac{M}{2}} = M - 1$;

(3-3). $C_i = 2i$; $2 \le i \le \frac{M}{2} - 1$;

(3-4). $C_{\frac{M}{2}+i} = C_{\frac{M}{2}-i} + 1$; $1 \le i \le \frac{M}{2} - 1$

(4). *Hardware Implementation Diagram*

Assume codeword C has N bits and dequantized number with 8 bits. The dequantizer in the decompressor can be implemented as following:



NOTE:

From the above diagram, it can be seen that the dequantization mapping has been dramatically reduced to less than a half of the original one.

(5). Example Table for the Range 64

For the following table: Rd=64, N=14;

Level index i	Decision Point Di	Recon. Level Qi	Codeword Ci
0	1	0	1110
1	4	3	000
2	7	6	0100
3	10	9	0110
4	13	11	1000
5	19	16	1010
6	27	23	1100
7	36	32	1111
8	44	41	1101
9	50	48	1011
10	53	53	1001
11	56	55	0111
12	59	58	0101
13	62	61	001

HD MPEG VIDEO DECODER CONTENTS

HD MPEG VIDEO DECODER

MAIN SPECIFICATION

Thomson Consumer Electronics, Inc.
Indianapolis, Indiana, USA

Revision No. 2.3

THESE DRAWINGS AND SPECIFICATIONS ARE THE PROPERTY OF THOMSON CONSUMER ELECTRONICS, INC. AND SHALL NOT BE REPRODUCED OR COPIED OR USED AS THE BASIS FOR THE MANUFACTURE OR SALE OF APPARATUS OR DEVICES WITHOUT WRITTEN PERMISSION.

TCE PROPRIETARY AND CONFIDENTIAL

TABLE OF CONTENTS

Appendices

1. INTRODUCTION

This specification delineates the electrical, mechanical, reliability, and product assurance requirements for a high definition MPEG decoder (HD-MPEG) IC. The HD-MPEG IC, along with external commodity memory devices and an external microcontroller, provides the necessary circuits for decoding and processing of MPEG compressed high definition image sequences.

This IC supports two display inferface modes:
a.) interface to a triple DAC, in which all display timing is generated by the HD video decoder IC.
b.) interface to an external NTSC encoder, in which the encoder provides the raster timing

2. TOP LEVEL DESCRIPTION

2.1. MPEG Video Decompression Module.

The HD-MPEG IC contains a module which decodes MPEG1 and MPEG2 video bit streams as described in Appendix A. A bit stream may correspond to a high definition video image sequence as specified in the Grand Alliance specification for US High Definition terrestrial television broadcast. A total pixel processing throughput rate of 94 Mpixels/sec (Y and C in 4:2:0 format) is required.

Reduced memory (memory compress/decompress) modes of operation are provided by the HD-MPEG IC. These modes allow significant reduction in the amount of memory required to decode HD image sequences by compressing video frames to be stored in memory and/or horizontally filtering and decimating pixel data within the decoding loop. These techniques are described in appendix G, Memory Reduction.

2.2. Applications Bus Interface

This functional block interfaces the IC to a parallel host bus in the application. The host bus interface provides a path for the input of compressed data as well as a bi-directional data path for controlling the IC. The interface is described in section 3.2 and Appendix B, Applications Bus.

2.3. Display Processor

This functional block provides a path for decoded pixels from the MPEG decompression module to an external display device. A programmable raster generator creates horizontal and vertical drive signals which can be used to synchronize an external display device.

Vertical resampling filters are provided which convert decoded 4:2:0 format pixels to 4:2:2 format for display. Horizontal and vertical resampling filters are provided to convert the decoded image resolution to a common resolution format for display.

The interface and digital signal processing are described in section 3.3 and appendix C, Display Processing.

2.4. On Screen Display (OSD)

This IC will provide a bit mapped on screen display capability. The ability to overlay any portion of the display while simultaneously decoding video must be provided.

The OSD module supports 2, or 4 bits per pixel which allows for selection from 4 or 16 palette registers. The OSD module also supports a 1 bit per pixel mode, in which a 4-bit foreground and 4-bit background index pointer references one of 16 palette registers for foreground and background color. Each palette register can be loaded with any color including transparent. The ability to mix video and OSD is also provided using a 4 bit weighting value.

Three resolution modes are provided. In full resolution mode 2 or 4 bits per pixel are used; in half resolution mode 2 or 4 bits per horizontal pixel pair are used (the odd and even field OSD pointers can also be the same which gives half resolution in the vertical direction). In one-third resolution mode, 2 or 4 bits per pixel triplette are used. Note that the one-third resolution mode is not compatible with the 8-bit output format (multiplexed Y,Cr,Cb). A special encoding mode is also available which uses one bit per pixel to switch between two previously defined palettes using a fixed length code.

The control microprocessor supplies the appropriate bit maps and control information. In addition a high speed memory block copy function is provided which improves the speed in which the external micro controller can create the desired bit maps.

The system is described in appendix D, On Screen Display.

2.5. External Memory Interface

This interface couples the MPEG2 decoder IC with up to 128 Mbits of external memory. Adequate bandwidth is provided to support decoding and display of high definition image formats and DSS formats as specified in Appendix A.

The HD-MPEG decoder IC generates all necessary control signals for interface to the external memories which can be of the synchronous graphics (SGRAM) or synchronous (SDRAM) type.

The width of the interface is 64 bits wide. Some details of the memory interface are given in Appendix L.

The memory is partitioned by values written into local memory controller registers by a host controller, using the applications bus. This memory partitioning designates memory locations for the compressed data bit buffer, OSD bit maps, and frame store buffers for MPEG decoding and display processing.

2.6. D1 Video Input

The HD-MPEG IC will be used in applications where standard definition video signals will be received and processed via an analog channel. A D1 Video pixel interface is provided which allows an input path for digitized component NTSC video signals in CCIR601 format conforming to the parallel "D1" input specification. All the display processing capabilities including horizontal and vertical sample rate conversion and OSD overlay can be applied to the input video before output.

2.7. Additional Features

This IC provides a number of additional features beyond a generic MPEG2 video decoder. These features include:

2.7.1. Error Concealment

A means for graceful handling of missing or erroneous bit stream data must be supported. The specific error concealment techniques to be used are described in section 3.6.

2.7.2. User Data

A means for making available specific user data from the received bit stream to the control microprocessor must be provided.

2.7.3. PES Layer Decoding

A start code aligned PES layer may be included as part of the compressed data input. The first byte following a PES header will be a video start code.

3. FUNCTIONAL DESCRIPTION

3.1. MPEG Processing

This IC provides the necessary processing capability to properly decode those bit streams making use of the compression techniques specified by MPEG1 (excluding "D" pictures) and MPEG2 main profile. Adequate processing throughput must be provided such that YCrCb pixel rates up to 94 Mpixels/sec can be decoded. The following are examples of image sequences which must be decoded by the HD-MPEG decoder IC.

Horizontal Resolutions	Vertical Resolutions	Frame Rate [Hz]	Syntax
1920	1080	30, 29.97, 24, 23.976	MPEG2 main profile
1280	720	60, 59.94, 30, 29.97, 24, 23.976	MPEG2 main profile
720 to 352, modulo 16	480	60, 59.94, 30, 29.97, 24, 23.976	MPEG1, MPEG2 main profile
720 to 352, modulo 16	240	60, 59.94, 30, 29.97, 24, 23.976	MPEG1, MPEG2 main profile

Note it is anticipated ALL resolution formats conforming to the following constraints will be decodable:

YCrCb coded pixel rate (HxVxFx1.5)	<= 94 Mpixels/sec (244,800 macroblocks/sec)
Compressed data rate	<= 80 Mbits/sec
YCrCb format	4:2:0
Syntax	MPEG1 or MPEG2 main profile
frame width	TBD (at least 1920 luma pixels or 120 Macroblocks)
frame size	<= 8160 macroblocks
bit buffer size	constrained only by available memory

3.2. Applications Bus Interface

The applications interface provides a communication path from a control microcomputer to the HD-MPEG video IC and also provides an input for the compressed video bit stream. This is an 8 bit parallel interface as described in Appendix B, Applications Bus.

This interface carries at least the following data:

Data to the HD-MPEG IC:

1. MPEG data stream
2. 16x9_Display control
3. Display_start control
4. Video output blank (to black when display is not enabled)

5. Disable_vbv operation control
6. OSD data stream
7. OSD control (See Appendix D: OSD)
8. Any other required setup information

Data from HD-MPEG IC

1. User data as specified above
2. MPEG parameters as specified above

Interrupt:

The host interface also provides an output line to interrupt the control microcomputer. The interrupt may be used to signal the following:

1. Start code detection
2. VLD error detection
3. Vertical sync detection
4. Bit buffer over- and under-flow
5. Bit buffer threshold for startup

Compressed Data Rate:

The HD-MPEG IC must support an average input compressed data rate of 80 Mbits/sec, over any interval of time spanning 188us or longer.

3.3. Display processor interface

The display processor interface performs a number of tasks associated with displaying 4:2:0 format decoded YCrCb pixels on a display device. The HD-MPEG IC can act as the master for the display or as a slave device.

3.3.1. Programmable raster generator

The HD-MPEG IC must have a flexible programmable raster generator which can be used to generate virtually any raster format. The following raster parameters must be programmable:

clocks_per_line	specifies how many display clock cycles are present in each line
half_lines_per_vertical	specifies how many output lines are present in each frame
HDO	specifies the delay of the generated horizontal drive pulse relative to H_reset
HDS	specifies end of generated horizontal drive pulse relative to H_reset
VDO	specifies delay of the generated vertical drive pulse relative to V_reset
'DS	specifies end of generated vertical drive pulse relative to V_reset
XDO	specifies delay of horizontal pixel output relative to H_reset
XDS	specifies end of horizontal pixel output relative to H_reset
YDO	specifies delay of vertical line output relative to V_reset
YDS	specifies end of vertical line output relative to V_reset

If half_lines_per_vertical is an odd number, the generated raster will be interlaced; if half_lines_per_vertical is an even number, the generated raster will be progressive.

The programming parameters must be latched by vertical sync which allows the control micro the ability to adjust them at a field rate.

H_reset and V_reset are virtual signals which occur when the pixel counter or half line counter are reset. When V_reset occurs it occurs synchronously with H_reset. These signals are generated internally due to the action of the pixel and half line counters; however provision must be provided to establish the phase (or gen-lock) these signals to an external signal if desired. The programming parameters defined above give the ability to establish a raster with arbitrary horizontal and vertical phase relative to an internal or external synchronizing signal.

3.3.2. Digital Signal Processing

The display processor converts decoded 4:2:0 format pixels to a raster format suitable for display. To ease display requirements the display interface converts the decoded image format into a common display format using horizontal and vertical resampling filters. The required filtering is setup in the HD-MPEG IC based in information found in the sequence header and knowledge of the type of display device in the system.

3.3.2.1. Horizontal Sample Rate Converter

A horizontal sample rate converter can be programmed to up convert the number horizontal output pixels. Virtually any up conversion factor is programmable.

3.3.2.2. Vertical Sample Rate Converter (down conversion)

A 3 tap vertical filter is used to sample rate the image vertically if desired. It is anticipated vertical sample rate conversion will be used to convert decoded 720 line progressive image sequences to 1080 line interlaced sequences. The vertical sample rate converter should be programmable to provide a wide range of conversion factors; however only a few ratios will be optimized. They include:

1:1.33	1080i from 720p (i.e. 540 field lines from 720 frame lines)
1:2.25	480i from 1080i (i.e. 240 field lines from 540 field lines)
1:3	480i from 720p (i.e. 240 field lines from 720 frame lines)

A 3 tap filter is used for the conversion.

3.3.2.3. Vertical Up Conversion (line doubler)

For display of SD video (either D1 input or MPEG decoded) on an HD raster a de-interlace algorithm is used to reduce the artifacts caused by displaying a 1H video signal on a 2H raster. This circuitry takes advantage of the idle memory resources available to the HD-MPEG IC when HD images are not being decoded.

3.3.3. Pan and scan:

The ability to start output display with a pixel other than the first is provided. This along with the resampling function provided with the horizontal filter can be used to provide pan and scan functionality. Information for 16x9 images consists of a signed 12 bit number representing, in quarter pixel units, the offset from the nominal central 4x3 area where the left edge should be located. The operation of Pan_and_Scan is to add the received differential value to the default value which equals Horizontal_size/8 (see Appendix: User Data).

If the source specified by the MPEG2 data stream is 16x9 and the display is 16x9, then the full pixel map is passed to the display. However, if the aspect ratio indicated in the MPEG2 data stream is 16x9 and the display is 4x3, then the pan and scan information is used such that only a 4x3 aspect ratio display is passed to the display. That is 544 pixels of a 720 pixel line, or 352 pixels of a 480 pixel line, are passed to the display.

A description of the desired interpolation filters and the timing of the interface signals are included in Appendix C: Display Processing.

3.4. On Screen Display (OSD)

This IC will provide an On Screen display capability. A bit mapped OSD will be implemented. The display will provide the ability to write over any part of the video display.

The preferred implementation is to use four bits per pixel with sixteen palette registers. The palette registers can be set to any color value or to a transparency mode which will allow video to pass when the register is selected.

Note the OSD is always created for the programmed raster type; OSD never passes through the horizontal or vertical resampling filters. The control microcomputer will supply the appropriate bit map taking into account the display raster parameters and the OSD resolution mode.

Block_Copy is the ability to copy the contents of one area of the external memory to another area. This feature can be used by the external micro controller to speed creation of OSD bit maps.

Such an OSD system is described in Appendix B: On Screen Display.

3.5. Memory requirements:

The HD-MPEG IC must provide all the address and control signals to operate with synchronous DRAMs and SGRAMs. An addressable memory space of up to 128 Mbits must be supported. An external bi-directional data bus of 64 bits must be available.

3.6. Error concealment

The following describes the actions which the HD-MPEG decoder IC should perform in various error conditions:

1. Sequence Header:
The sequence header information shall be double buffered such that existing values will not be replaced by new values until all of the new values have been received correctly. That is, present values will not be changed if a media error code is received in a sequence header or if a sequence header is otherwise determined to be in error.

2. GOP Header:
GOP header is not replaced until a new GOP header is received correctly.

3. Media error code (sequence_error_code in MPEG2 syntax):

The transport IC inserts a media error code (00 00 01 B4) into the video bit stream when an error (or missing data) is detected. This error code is detected and processed by the MPEG video decoder in the following steps:

1. On media error code, store current Macro Block Address (MBA) and search for the next MPEG start code.

2. If the next MPEG start code is a picture start code or above, the rest of the image is lost and the rest of the frame needs to be replaced. After replacement, decode the next MPEG word. If the next MPEG word is not a media error code, resume normal processing. Otherwise reset the MBA to the first MB of new frame and search for the next MPEG start code.

3. If the next MPEG start code is not a picture start code or above, decode the MBA and compare with the old stored MBA. If the difference is greater than zero, assume that lost MBs are within the same frame and conceal all macroblocks between the two MBAs. If the difference is smaller than zero (MBs are lost across frames), conceal all remaining macroblocks in the present frame, and search for the next picture start code or above concealing all lost frames. Figure 2 illustrates this process in more detail.

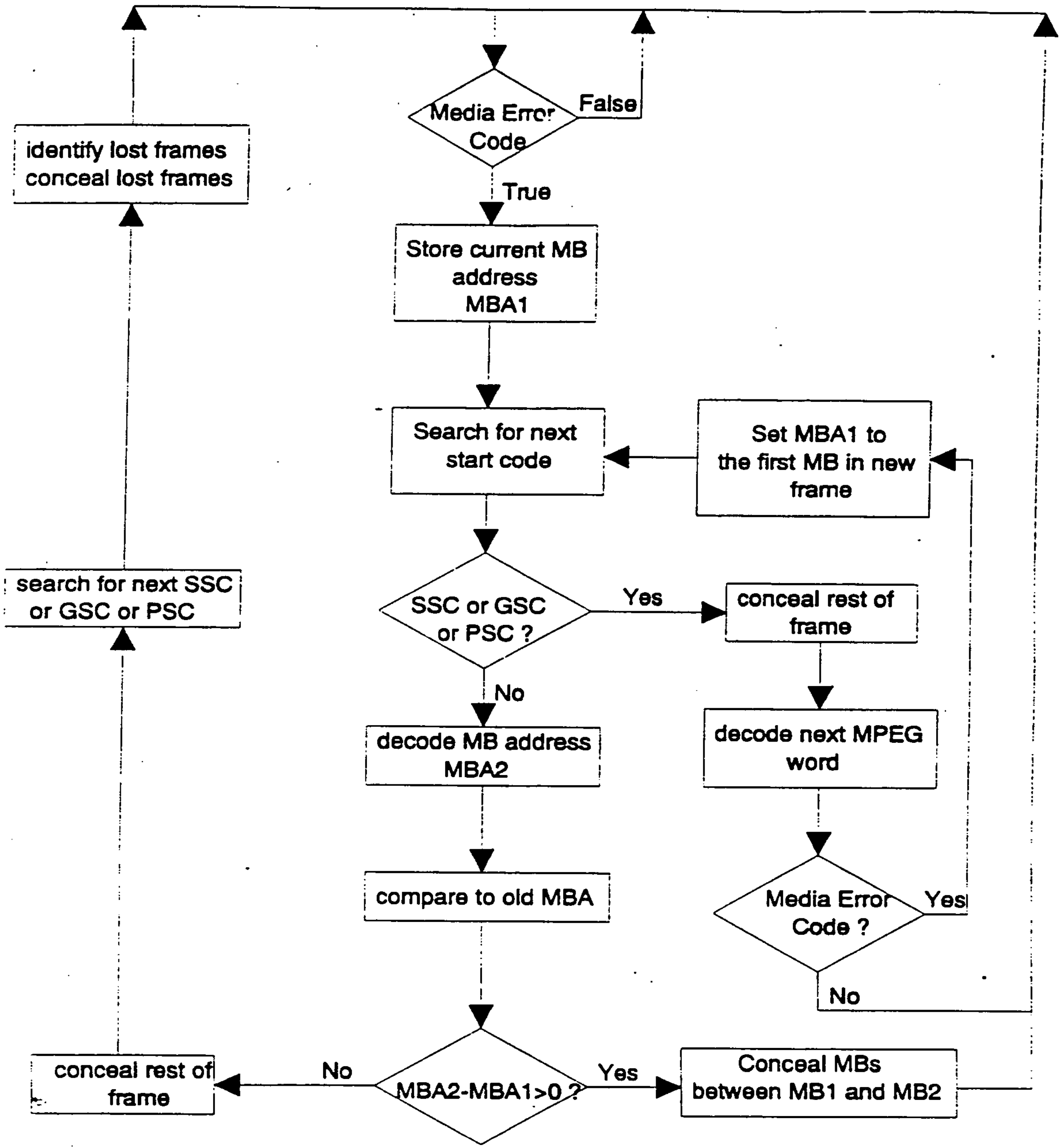


Figure 2: Logical Chart for Macroblock Level Error Concealment in an MPEG decoder.

After the error area is identified, all impacted macroblocks (MBs) are concealed by appropriate replacement of MB data with MB type, motion vectors (MV), and all other attributes from vertically adjacent MBs.

The concealment method is as follows:

During normal operation, store MB type, motion compensation (MC) type and all MVs. MB type in replaced macroblocks is reset to use motion compensation alone in reconstruction (no error signal processing). For MPEG2 field/frame mode this translates into storing previous raster of MB type, MC type and MVs to support this concealment logic.

I frame: lost MBs are replaced with co-located ones from the last decoded anchor frame.

P frame: lost MBs are replaced with corresponding ones from the last decoded anchor frame with motion compensation. Special cases:

- If the lost MBs are in the first row, the are replaced with co-located ones from the last decoded anchor frame.

- If the lost MBs are not in the first row, they are replaced with corresponding MBs from the previous frame using motion compensation. The MB type and MC type are decided by the corresponding top neighbors of MB. That is, if the top MB is intra, the lost MB is replaced with a co-located one from the previous frame. If the top is motion compensated, the lost one is replaced with motion compensated MB. If the top MB is in field mode, the lost one is replaced with a MB in the same mode, motion compensated.

- The lost information is estimated from vertically adjacent neighbors. If the motion vector points outside of the picture, it is forced to zero.

B frame: lost MBs are replaced with corresponding ones from the last decoded anchor frame with motion compensation. Special cases:

- If the lost MBs are in the first row, the are replaced with co-located ones from the last decoded anchor frame.

- If the lost MBs are not in the first row, they are replaced with corresponding MBs from the previous frame using motion compensation. The MB type and MC type are decided by the corresponding top neighbors of MB. That is, if the top MB is intra, the lost MB is replaced with a co-located one from the previous frame. If the top is motion compensated, the lost one is replaced with motion compensated MB in the same mode. If the top MB is in field mode, the lost one is replaced with a MB in the same mode, motion compensated.

- The lost information is estimated from vertically adjacent neighbors. If the motion vector points outside of the picture, it is forced to zero.

3.7. User Data

Appendix E: User Data describes specific user data to be made available to the control microcomputer. Appropriate means in the MPEG2 IC are available for control of:

1. Pan and Scan
2. Field display code.
3. Presentation Time
4. Horizontal_size
5. Vertical_size
6. Pel Aspect ratio

The remaining user data is made available to the control microcomputer via the host/data bus along with other parameters delivered in the bit stream. These include:

7. Closed Caption
8. Closed_GOP
9. Broken_link
10. Open_GOP

3.8. Nominal Operating Conditions

The HD-MPEG decoder IC operates with a single supply voltage of 3.3 v +/- 10%.

A number of internal processing clocks are used by the HD-MPEG decoder IC. The display system must support a primary clock frequency of up to 81 MHz. It is anticipated the MPEG decoder module must support a primary clock frequency of up to 54 MHz. The HD-MPEG decoder IC should be capable of operation with asynchronous display and decoding clocks or with clocks generated from a common time base.

4. APPLICABLE STANDARD DOCUMENTS.

The following documents and standard specifications contain general requirements which the supplier must meet.

General IC Approval Requirements, 50895020
Effects of Soldering, 40438960
MIL-STD-883C, Method 1014.5 Seal, condition D
MIL-STD-883C, Method 2004.4 Lead Integrity, conditions A & B2
MIL-STD-883C, Method 2010,7 Internal Visual (Monolithic), Cond. B
MIL-STD-883C, Method 2012.5 Radiography
MIL-STD-883C, Method 2018.1 SEM Inspection of Metallization
MIL-STD-883C, Method 2011.4 Wire Pull, condition D
MIL-STD-883C, Method 1010.5 Temperature Cycling, condition C
MIL-STD-883R, Method 3015.1 Electrostatic Discharge

5. PHYSICAL AND THERMAL REQUIREMENTS.

This device shall dissipate less than 2 watts. The maximum junction temperature shall not exceed 90°C while operating in a 50°C ambient. Total number of pins on the IC shall be 208.

5.1. Physical Requirements.

This IC must be packaged in a 208 pin PQFP-package in accordance with Figure 3.

Pin Assignments: To Be Determined

Pin Number	Name	Description	Type	Total
1-208	HD-MPEG	high definition MPEG decoder IC		208
		Applications Bus Interface		22
TBD	A_nCS	chip select	input	1
TBD	A_R/nW	read/write control	input	1
TBD	A_Addr[7:0]	register address	input	8
TBD	A_Data[7:0]	register data and compressed data input	bidir	8
TBD	A_nWAIT	wait line	output	1
TBD	A_nREQ	request for input compressed data	output	1
TBD	A_nSTRB	input compressed data strobe	input	1
TBD	A_nIRQ	interrupt request	output	1
		D1 Video Interface		11
TBD	V_VSync	Input V sync signal	input	1
TBD	V_HSync	Input H sync signal	input	1
TBD	V_ClkIn	27 MHz pixel input clock	input	1
TBD	V_Data[7:0]	muxed YCrCb SD pixels	input	8
		Display Interface		28
TBD	D_Y/C[7:0]	luma output or multiplexed Y,Cr,Cb output	output	8
TBD	D_Cb[7:0]	component Cb output	output	8
TBD	D_Cr[7:0]	component Cr output	output	8
TBD	D_ClkOut	pixel clock out	output	1
TBD	D_Hdrive	horizontal display reference	bidir	1
TBD	D_Vdrive	vertical display reference or external B/nT	bidir	1
TBD	D_OSDactive	OSD presence indicator	output	1
		Memory Interface		84
TBD	M_ClkOut	memory data clock out	output	1
TBD	M_ClkIn	memory data clock in	input	1
TBD	M_nRAS	row address strobe control	output	1
TBD	M_nCAS	column address strobe control	output	1
TBD	M_nCS[1:0]	memory chip select	output	2
TBD	M_nWE	memory write enable not	output	1
TBD	M_Addr[11:0]	memory base address and bank	output	12
TBD	M_DQM	memory data qualifier mask	input	1
TBD	M_Data[63:0]	external memory data bus	bidir	64
		System Miscellaneous		62
TBD	S_nRESET	system reset	input	1
TBD	S_SClkIn	reference system clock input	input	1
TBD	S_Clk0	reference CLK0 input/output for LMC/Memory	bi-dir	1
TBD	S_Clk1	reference CLK1 input/output for VLD/Decode Pipe	bi-dir	1
TBD	S_Clk2	reference CLK2 input/output for Display	bi-dir	1
TBD	S_Clk3	reference CLK3 input/output for Decompress	bi-dir	1
TBD	S_TestEn	Test enable	input	1

TBD	S_DBusClkIn	D-Bus Test Clock	input	1
TBD	S_ScanEn	Scan enable	input	1
TBD	VCC_VCO1	Isolated +3.3 volt VCC supply for VCO1	input	1
TBD	GND_VCO1	Isolated GND supply for VCO1	input	1
TBD	VCC_VCO2	Isolated +3.3 volt VCC supply for VCO2	input	1
TBD	GND_VCO2	Isolated GND supply for VCO2	input	1
TBD	VCC	Main +3.3 volt VCC supply	input	24
TBD	GND	Main GND supply	input	24
TBD	VCC_DRAM	DRAM supply	input	1
TBD	GND_DRAM	DRAM supply	input	1
TBD	spare	undefined/available pins		0

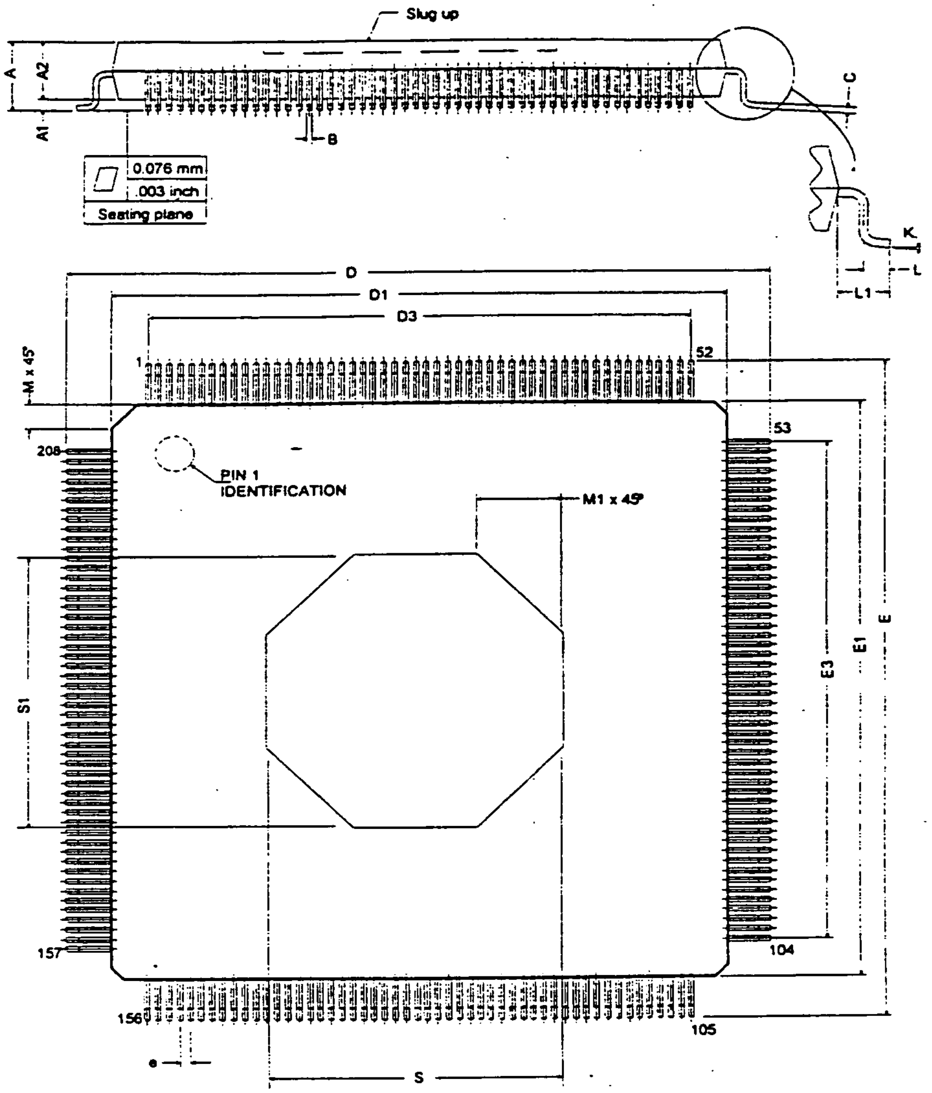


Figure 3: MPEG2 decoder IC package dimentions.

ref	A	A1	A2	B	c	D	D1	D3	e	E
typ			3.40	0.22		30.60	28.00	25.50	0.50	30.60
min		0.25	3.20	0.17	0.09					
max	4.07		3.60	0.27	0.20					

ref	E1	E3	L	L1	k	M	M1	S1	S2
typ	28.00	25.50	0.60	1.30		0.95	3.80	13.00	13.00
min			0.50		0 deg				
max			0.75		7 deg				

5.2. Thermal Requirements

Guaranteed storage temperature range: -50°C to +150°C
Guaranteed operating temperature range: 0°C to 70°C
Device junction temperature shall not exceed 90°C while operating in a 50°C ambient.

5.3. Maximum Ratings.

The device shall be capable of withstanding the following maximum conditions without permanent damage:

ESD: 2 KV pulses applied to all terminals as described in MIL-SRD-883R noted above.
Power dissipation shall be less than or equal to 2W.
Lead Temperature (10 seconds during soldering): 260°C.
Maximum clock frequency: [*to be specified*].

6. ELECTRICAL CHARACTERISTICS.

All electrical tests shall be performed at 25°C ambient, at nominal supply voltage, test circuits per fig.x -fig.x unless otherwise specified.

All logic inputs and outputs shall meet LV-TTL level requirements.

All application interface pins (host bus interface, display interface) shall be +5 volt tolerant and compatible with both TTL levels for 5v devices and LVTTL levels for interface with 3.3v devices.

All memory interface pins shall be compatible with the intended commodity memory devices.

APPLICATIONS BUS						
I/O Signal	voltage levels, 5v tolerant	capacitance	leakage current	max VOL @ IOL	min VOH @ IOH	Rise/Fall Time @ 100 pF
A_Addr[7:0]	LVTTL inputs	10 pF input	+/- 1 uA	NA	NA	NA
A_Data[7:0]	LVTTL I/O	10 pF	+/- 5 uA	0.4 V @ 8 mA	2.4 V @ 8 mA	5.6 nsec/ 9.5 nsec
A_nSTRB	LVTTL input	10 pF input	+/- 1 uA	NA	NA	NA
A_nREQ	LVTTL output	10 pF	+/- 5 uA	0.4 V @ 8 mA	2.4 V @ 8 mA	5.6 nsec/ 9.5 nsec
A_nCS	LVTTL input	10 pF input	+/- 1 uA	NA	NA	NA
A_R/nW	LVTTL input	10 pF input	+/- 1 uA	NA	NA	NA
A_nIRQ	LVTTL output	10 pF	+/- 5 uA	0.4 V @ 8 mA	2.4 V @ 8 mA	5.6 nsec/ 9.5 nsec
A_nWAIT	LVTTL TS output	10 pF	+/- 5 uA	0.4 V @ 8 mA	2.4 V @ 8 mA	5.6 nsec / 9.5 nsec

D1 VIDEO INTERFACE						
I/O Signal	voltage levels, 5v tolerant	capacitance	leakage current	max VOL @ IOL	min VOH @ IOH	Rise/Fall Time @ 100 pF
V_Data[7:0]	LVTTL inputs	10 pF	+/- 5 uA	NA	NA	NA
V_Clkin	LVTTL input	10 pF	+/- 1 uA	NA	NA	NA
V_VSync	LVTTL input	10 pF	+/- 1 uA	NA	NA	NA
V_HSync	LVTTL input	10 pF	+/- 1 uA	NA	NA	NA

DISPLAY INTERFACE						
I/O Signal	voltage levels, 5v tolerant	capacitance	leakage current	max VOL @ IOL	min VOH @ IOH	Rise/Fall Time @ 100 pF
YC[7:0]	LVTTL outputs	10 pF	+/- 5 uA	0.4 V @ 8 mA	2.4 V @ 8 mA	5.6 nsec/ 9.5 nsec
PIXCLK	LVTTL output	10 pF	+/- 1 uA	NA	NA	NA
B/nT	LVTTL input	10 pF	+/- 1 uA	NA	NA	NA
HSYNC	LVTTL input	10 pF	+/- 1 uA	NA	NA	NA

DRAM INTERFACE						
I/O Signal	voltage levels	capacitance	leakage current	max VOL @ IOL	min VOH @ IOH	Rise/Fall Time @ 100 pF
M_Addr[13:0]	LVTTL output	10 pF	+/- 5 uA	0.4 V @ 8 mA	2.4 V @ 8 mA	5.6 nsec/ 9.5 nsec
M_Data[63:0]	LVTTL I/O	10 pF	+/- 5 uA	0.4 V @ 8 mA	2.4 V @ 8 mA	5.6 nsec/ 9.5 nsec
nRAS	LVTTL output	10 pF	+/- 5 uA	0.4 V @ 8 mA	2.4 V @ 8 mA	5.6 nsec/ 9.5 nsec
nCAS[U:L]	LVTTL outputs	10 pF	+/- 5 uA	0.4 V @ 8 mA	2.4 V @ 8 mA	5.6 nsec/ 9.5 nsec
nOE	LVTTL output	10 pF	+/- 5 uA	0.4 V @ 8 mA	2.4 V @ 8 mA	5.6 nsec/ 9.5 nsec
nWE	LVTTL output	10 pF	+/- 5 uA	0.4 V @ 8 mA	2.4 V @ 8 mA	5.6 nsec/ 9.5 nsec

SYSTEM INTERFACE						
I/O Signal	voltage levels	capacitance	leakage current	max VOL @ IOL	min VOH @ IOH	Rise/Fall Time @ 100 pF
	LVTTL output	10 pF	+/- 5 uA	0.4 V @ 8 mA	2.4 V @ 8 mA	5.6 nsec/ 9.5 nsec
	LVTTL I/O	10 pF	+/- 5 uA	0.4 V @ 8 mA	2.4 V @ 8 mA	5.6 nsec/ 9.5 nsec
	LVTTL output	10 pF	+/- 5 uA	0.4 V @ 8 mA	2.4 V @ 8 mA	5.6 nsec/ 9.5 nsec
	LVTTL outputs	10 pF	+/- 5 uA	0.4 V @ 8 mA	2.4 V @ 8 mA	5.6 nsec/ 9.5 nsec
	LVTTL output	10 pF	+/- 5 uA	0.4 V @ 8 mA	2.4 V @ 8 mA	5.6 nsec/ 9.5 nsec
nRESET	Shmitt input LVTTL compatible	10 pF	+/- 1 uA	NA	NA	NA

FIGURE 4. TEST CIRCUIT(S)

7. REFERENCES:

[1] ISO 11172-1 and 11172-2 (MPEG1 Systems, Video)
[2] ISO 13818-1 and 13818-2 (MPEG2 Systems, Video)

8. MEMORY UTILIZATION:

This section identifies the amount of external memory consumed based upon decoded frame size and compression modes used.

In the following three tables:

Frames per second rate of 60 includes both 60 and 59.94.
Frames per second rate of 24 includes both 24 and 23.98.
Frames per second rate of 30 includes both 30 and 29.97.
Horizontal pixel values of 720 also include 704

8.1. Set-Top Application

Application memory is 33,554,432 bits. Display is 480i active lines (29.97/30 fps).
OSD memory = 720x480x4 = 1,382,400

Input Format HxV, frames/s	Bit Buffer	Anchor Frames	Display or B-Frame	OSD Memory	Free Memory	LMU Active	Compr. Mode	Displayed pixels HxV,fps
352x240, 24, 30	1,875,000	2,027,520	1,013,760	1,382,400	27,255,752	no	off	720x480i, 30
352x480, 24, 30	1,875,000	4,055,040	2,027,520	1,382,400	25,214,472	no	off	720x480i, 30
544x480, 24, 30	1,875,000	6,266,880	3,133,440	1,382,400	22,279,112	no	off	720x480i, 30
640x480, 24, 30, 60p	9,500,000	7,372,400	3,686,400	1,382,400	11,613,232	no	off	720x480i, 30
720x480, 24, 30, 60p	9,500,000	8,294,400	4,147,200	1,382,400	10,230,432	no	off	720x480i, 30
1280x720p, 24, 30, 60	9,500,000	5,529,600	2,764,800	1,382,400	14,377,632	no	H/2, M/2	720x480i, 30
1280x720p, 24, 30, 60	9,500,000	14,745,600	7,372,800	1,382,400	553,632	no	2M/3	720x480i, 30
1920x1080, 24, 30	9,500,000	12,441,600	6,220,800	1,382,400	4,009,632	no	H/2, M/2	720x480i, 30

8.2. SD-Application

Application memory is 33,554,432 bits. Display is 480p active lines (59.94/60 fps).
= only for interlaced input formats. OSD memory = 720x480x4 = 1,382,400

Input Format HxV, frames/s	Bit Buffer	Anchor Frames	Display or B-Frame	OSD Memory	Free Memory	LMU Active	Compr. Mode	Displayed pixels H(alternate H)xV, fps
720x480i, 30 (D1)	0	0	6,220,800	1,382,400	25,951,232	yes*	off	720(960)x480p, 60
352x240, 24, 30	1,875,000	2,027,520	1,013,760	1,382,400	27,255,752	yes*	off	720(960)x480p, 60
352x480, 24, 30	1,875,000	4,055,040	2,027,520	1,382,400	25,214,472	yes*	off	720(960)x480p, 60
544x480, 24, 30	1,875,000	6,266,880	3,133,440	1,382,400	22,279,112	yes*	off	720(960)x480p, 60
640x480, 24, 30, 60p	9,500,000	7,372,400	3,686,400	1,382,400	11,613,232	yes*	off	720(960)x480p, 60
720x480, 24, 30, 60p	9,500,000	8,294,400	4,147,200	1,382,400	10,230,432	yes*	off	720(960)x480p, 60
1280x720p, 24, 30, 60	9,500,000	5,529,600	2,764,800	1,382,400	14,377,632	no	H/2, M/2	720(960)x480p, 60
1280x720p, 24, 30, 60	9,500,000	14,745,600	7,372,800	1,382,400	553,632	no	2M/3	960(1280)x480p, 60
1920x1080, 24, 30	9,500,000	12,441,600	6,220,800	1,382,400	4,009,632	no	H/2, M/2	720(960)x480p, 60

8.3. HD-Application

Application memory is 67,108,864 bits. Display is 1080i (29.97/30 fps), 540p (59.94/60), or 480p-zoom (59.94/60) ctive lines.
* = only for interlaced input formats. OSD memory = 960x540x4 = 2,073,600

Input Format HxV, frames/s	Bit Buffer	Anchor Frames	Display or B-Frame	OSD Memory	Free Memory	LMU Active	Compr. Mode	Displayed Pixels HxV(alternate V), fps
720x480i, 30 (D1)	0	0	6,220,800	2,073,600	58,814,464	yes*	off	1920x1080i, 30
352x240, 24, 30	1,875,000	2,027,520	1,013,760	2,073,600	60,118,984	yes*	off	1920x1080i, 30
352x480, 24, 30	1,875,000	4,055,040	2,027,520	2,073,600	57,077,704	yes*	off	1920x1080i, 30
544x480, 24, 30	1,875,000	6,266,880	3,133,440	2,073,600	53,757,944	yes*	off	1920x1080i, 30
640x480, 24, 30, 60p	9,500,000	7,372,400	3,686,400	2,073,600	44,476,464	yes*	off	1920x1080i, 30
720x480, 24, 30, 60p	9,500,000	8,294,400	4,147,200	2,073,600	43,093,664	yes*	off	1920x1080i, 30
1280x720p, 24, 30, 60	9,500,000	22,118,400	11,059,200	2,073,600	16,828,064	no	off	1920x1080i, 30
1920x1080, 24, 30	9,500,000	33,177,600	16,588,800	2,073,600	792,224	no	2M/3	1920x1080i, 30

8.4. HD-Full Memory

Application memory is 134,217,728 bits. Display is 1080i (29.97/30 fps), 540p (59.94/60), or 480p-zoom (59.94/60) active lines.
* = only for interlaced input formats. OSD memory = 960x540x4 = 2,073,600

Input Format HxV, frames/s	Bit Buffer	Anchor Frames	Display or B-Frame	OSD Memory	Free Memory	LMU Active	Compr. Mode	Displayed Pixels HxV(alternate V), fps
720x480i, 30 (D1)	0	0	6,220,800	2,073,600	125,923,328	yes*	off	1920x1080i, 30
352x240, 24, 30	1,875,000	2,027,520	1,013,760	2,073,600	127,227,848	yes*	off	1920x1080i, 30
352x480, 24, 30	1,875,000	4,055,040	2,027,520	2,073,600	124,186,568	yes*	off	1920x1080i, 30
544x480, 24, 30	1,875,000	6,266,880	3,133,440	2,073,600	120,868,808	yes*	off	1920x1080i, 30
640x480, 24, 30, 60p	9,500,000	7,372,400	3,686,400 -	2,073,600	111,585,328	yes*	off	1920x1080i, 30
720x480, 24, 30, 60p	9,500,000	8,294,400	4,147,200	2,073,600	110,202,528	yes*	off	1920x1080i, 30
1280x720p, 24, 30, 60	9,500,000	22,118,400	13,271,040	2,073,600	87,254,688	no	off	1920x1080i, 30
1920x1080, 24, 30	9,500,000	49,766,400	29,859,840	2,073,600	43,017,888	no	off	1920x1080i, 30

HD MPEG VIDEO DECODER

APPENDIX A

MPEG BITSTREAM CONSTRAINTS AND DECODE REQUIREMENTS

Thomson Consumer Electronics, Inc.
Indianapolis, Indiana, USA

Revision No. 2.1

THESE DRAWINGS AND SPECIFICATIONS ARE THE PROPERTY OF THOMSON CONSUMER ELECTRONICS, INC. AND SHALL NOT BE REPRODUCED OR COPIED OR USED AS THE BASIS FOR THE MANUFACTURE OR SALE OF APPARATUS OR DEVICES WITHOUT WRITTEN PERMISSION.

TCE PROPRIETARY AND CONFIDENTIAL

A1. PES LAYER CONSTRAINTS

A1.1. SCOPE

The HD-MPEG video decoder IC shall be capable of decoding the following video streams and systems packets:

a.) MPEG1 and MPEG2 video, with DSS syntax extentions and constraints.
b.) MPEG2 MP@ML video
c.) MPEG2 MP@HL video with ATSC/GA constraints
d.) An MPEG2 aligned PES packet containing any of the above MPEG2 streams.

The scope of this section covers item "d".

A1.2. PES Syntax

Packetized Elementary Stream syntax and semantics shall be used to encapsulate video elementary stream information. The Packetized Elementary Stream syntax is used to convey the Presentation Time-Stamp (PTS) and Decoding Time-Stamp (DTS) information required for decoding audio and video information with synchronism.

The PES Layer syntax and semantics shall be that as described within ISO/IEC 13818-1.

A1.3. PES Constraints (aligned)

This Section describes the coding constraints for this system layer.

Within the PES packet header, the following restrictions apply:

PES_scrambling_control shall be coded as '00'.
ESCR_flag shall be coded as '0'.
ES_rate_flag shall be coded as '0'.
PES_CRC_flag shall be coded as '0'.

Within the PES packet extension, the following restrictions apply.

PES_private_data_flag shall be coded as '0'.
pack_header_field_flag shall be coded as '0'.
program_packet_sequence_counter_flag shall be coded as '0'.
P-STD_buffer_flag shall be coded as '0'.

Additional Video PES constraints

Each PES packet shall begin with a video access unit, as defined in Section 2.1.1 of ISO/IEC 13818-1, which is aligned with the PES packet header. The first byte of a PES packet payload shall be the first byte of a video access unit. Each PES header shall contain a PTS. Additionally, it shall contain a DTS as appropriate. For terrestrial broadcast, the PES packet shall not contain more than one coded video frame, and shall be void of video picture data only when transmitted in conjunction with the discontinuity_indicator to signal that the continuity_counter may be discontinuous.

Within the PES packet header, the following restrictions apply:

The PES_packet_length shall be coded as '0x0000'.
data_alignment_indicator shall be coded as '1'.

The contents of the PES packet header shall be accessible to the host controller, as read after the bit buffer.

A1.4. PES Constraints (Unaligned)

The PES contains no constraints beyond those identified in ISO/IEC 13818-1. The contents of the PES packet header shall be accessible to the host controller, as read before the bit buffer. This header is buffered independently of compressed data in the bit buffer.

A2. ATSC/GA CONSTRAINTS

A2.1. SCOPE

The HD-MPEG video decoder IC shall be capable of decoding

a.) MPEG1 and MPEG2 with DSS syntax and constraints.
b.) MPEG2 MP@ML
c.) MPEG2 MP@HL with ATSC/GA constraints
d.) MPEG2 video that may be contained within an ISO13818-1 aligned PES packet.

The scope of this section covers item "c".

A2.2. Possible video inputs

While not required by this standard, there are certain television production standards, shown in Table A1, that define video formats that relate to compression formats specified by this standard.

Table A1 Standardized Video Input Formats

Video standard	Active lines	Active samples/ line
SMPTE 274M	1080	1920
SMPTE S17.392	720	1280
ITU-R BT.601-4	483	720

The compression formats may be derived from one or more appropriate video input formats. It may be anticipated that additional video production standards will be developed in the future that extend the number of possible input formats.

A2.3. Source coding specification

The ATV video compression algorithm shall conform to the Main Profile syntax of ISO/IEC 13818-2. The allowable parameters shall be bounded by the upper limits specified for the Main Profile at High Level.[1] Additionally, ATV bit streams shall meet the constraints and specifications described in Sections 5.1 and 5.2.

A2.3.1. Constraints with respect to MPEG-2 MP@HL

The following tables list the allowed values for each of the ISO/IEC 13818-2 syntactic elements which are restricted beyond the limits imposed by MP@HL.

In these tables conventional numbers denote decimal values, numbers preceded by **0x** are to be interpreted as hexadecimal values and numbers within single quotes (e.g., '10010100') are to be interpreted as a string of binary digits.

A2.3.1.1. Sequence header constraints

Table 2 identifies parameters in the sequence header of a bit stream that shall be constrained by the video subsystem and lists the allowed values for each.

[1] See ISO/IEC 13818-2, Section 8 for more information regarding profiles and levels.

Table 2 Sequence Header Constraints

Sequence header syntactic element	Allowed value
horizontal_size_value	see Table 3
vertical_size_value	see Table 3
aspect_ratio_information	see Table 3
frame_rate_code	see Table 3
bit_rate_value (≤ 19.4 Mbps)	≤ 48500
bit_rate_value (≤ 38.8 Mbps)	≤ 97000
vbv_buffer_size_value	≤ 488

The allowable values for the field bit_rate_value are application dependent. In the primary application of terrestrial broadcast, this field shall correspond to a bit rate which is less than or equal to 19.4 Mbps. In the high data rate mode, the corresponding bit rate is less than or equal to 38.8 Mbps.

A2.3.1.2. Compression format constraints

Table A3 lists the allowed compression formats.

Table A3 Compression Format Constraints

vertical_size_value	horizontal_size_value	aspect_ratio_information	frame_rate_code	progressive_sequence
1080[2]	1920	1,3	1,2,4,5	1
			4,5	0
720	1280	1,3	1,2,4,5,7,8	1
480	704	2,3	1,2,4,5,7,8	1
			4,5	0
	640	1,2	1,2,4,5,7,8	1
			4,5	0

Legend for MPEG-2 coded values in Table A3
aspect_ratio_information 1 = square samples 2 = 4:3 display aspect ratio 3 = 16:9 display aspect ratio
frame_rate_code 1 = 23.976 Hz 2 = 24 Hz 4 = 29.97 Hz 5 = 30 Hz 7 = 59.94 Hz 8 = 60 Hz
progressive_sequence 0 = interlaced scan 1 = progressive scan

A2.3.1.3. Sequence extension constraints

Table A4 identifies parameters in the sequence extension part of a bit stream that shall be constrained by the video subsystem and lists the allowed values for each. A sequence_extension structure is required to be present after every sequence_header structure.

Table A4 Sequence Extension Constraints

Sequence extension syntactic element	Allowed values
progressive_sequence	see Table 3
profile_and_level_indication	see Note
chroma_format	'01'

[2] Note that 1088 lines are actually coded in order to satisfy the MPEG-2 requirement that the coded vertical size be a multiple of 16 (progressive scan) or 32 (interlaced scan).

horizontal_size_extension	'00'
vertical_size_extension	'00'
bit_rate_extension	'0000 0000 0000'
vbv_buffer_size_extension	'0000 0000'
frame_rate_extension_n	'00'
frame_rate_extension_d	'0000 0'

Note: The profile_and_level_indication field shall indicate the lowest profile and level defined in ISO/IEC 13818-2, Section 8, that is consistent with the parameters of the video elementary stream.

A2.3.1.4. Sequence display extension constraints

Table A5 identifies parameters in the sequence display extension part of a bit stream that shall be constrained by the video subsystem and lists the allowed values for each.

Table A5 Sequence Display Extension Constraints

Sequence display extension syntactic element	Allowed values
video_format	'000'

The preferred and default values for color_primaries, transfer_characteristics, and matrix_coefficients are defined to be SMPTE 274M[3] (value 0x01 in all three cases). While all values described by MPEG-2 are allowed in the transmitted bit stream, it is noted that SMPTE 170M values (0x06 in all three cases) will be the most likely alternate in common use.

A2.3.1.5. Picture header constraints

In all cases other than when vbv_delay has the value 0xFFFF, the value of vbv_delay shall be constrained as follows:

vbv_delay <= 45000

A2.3.2. Bit stream specifications beyond MPEG-2

This section covers the extension and user data part of the video syntax. These data are inserted at the sequence, GOP, and picture level. The syntax used for the insertion of closed captioning in picture user data is described.[4]

A2.3.2.1. Picture extension and user data syntax

Table A6 describes the syntax used for picture extension and user data.

Table A6 Picture Extension and User Data Syntax

	No. of bits	Mnemonic
extension_and_user_data(2) {		

[3] At some point in the future, the color gamut may be extended by allowing negative values of RGB and defining the transfer characteristics for negative RGB values.

[4] In order to decode the user data, the decoder should properly recognize the 32-bit ATSC registration identifier at the PSI stream level (see ISO/IEC 13818-1).

while ((nextbits( ) = = extension_start_code) || (nextbits() = = user_data_start_code)) {		
if (nextbits() = = extension_start_code)		
extension_data(2)		
if (nextbits() = = user_data_start_code)		
user_data(2)		
}		
}		

A2.3.2.2. Picture user data syntax

Table A7 describes the picture user data syntax.

Table A7 Picture User Data Syntax[5]

	No. of bits	Mnemonic
user_data( ) {		
user_data_start_code	32	bslbf
ATSC_identifier	32	bslbf
user_data_type_code	8	uimsbf
if (user_data_type_code = = '0x03') {		
process_em_data_flag	1	bslbf
process_cc_data_flag	1	bslbf
additional_data_flag	1	bslbf
cc_count	5	uimsbf
em_data	8	bslbf
for (i=0 ; i < cc_count ; i++) {		
marker_bits	5	'1111 1'
cc_valid	1	bslbf
cc_type	2	bslbf
cc_data_1	8	bslbf
cc_data_2	8	bslbf
}		
marker_bits	8	'1111 1111'
if (additional_data_flag) {		
while(nextbits() != '0000 0000 0000 0000 0000 0001') {		
additional_user_data	8	

[5] Shaded cells in this table indicate syntactic and semantic additions to the ISO/IEC 13818-2 standard.

}		
}		
}		
next_start_code()		
}		

A2.3.2.3. Picture user data semantics

user_data_start_code — This is set to 0x0000 01B2.

ATSC_identifier — This is a 32 bit code that indicates that the video user data conforms to this specification. The value ATSC_identifier shall be 0x4741 3934.

user_data_type_code — The 8-bit code is set to 0x03.

process_em_data_flag — This flag is set to indicate whether it is necessary to process the em_data. If it is set to 1, the em_data has to be parsed and its meaning has to be processed. When it is set to 0, the em_data can be discarded.

process_cc_data_flag — This flag is set to indicate whether it is necessary to process the cc_data. If it is set to 1, the cc_data has to be parsed and its meaning has to be processed. When it is set to 0, the cc_data can be discarded.

additional_data_flag — This flag is set to 1 to indicate the presence of additional user data.

cc_count — This 5-bit integer indicates the number of closed caption constructs following this field. It can have values 0 through 31. The value of cc_count shall be set according to the frame rate and coded picture structure (field or frame) such that a fixed bandwidth of 9600 bits per second is maintained for the closed caption payload data. Sixteen (16) bits of closed caption payload data are carried in each pair of the fields cc_data_1 and cc_data_2.

em_data — Eight bits for representing emergency message.[6]

cc_valid — This flag is set to '1' to indicate that the two closed caption data bytes that follow are valid. If set to '0' the two data bytes are invalid.

cc_type — Denotes the type of the two closed caption data bytes that follow.[7]

cc_data_1 — The first byte of a closed caption data pair.

cc_data_2 — The second byte of a closed caption data pair.

additional_user_data — Any further demand for picture user data could be met by defining this part of the bit stream.

[6] Syntax and semantics to be specified by EIA.

[7] EIA, *Recommended Practice for Advanced Television Closed Captioning*, draft, July 1, 1994.

A3. MPEG2 MAIN PROFILE VIDEO BIT STREAM SYNTAX (13818-2)

A3.1. SCOPE

The HD-MPEG video decoder IC shall be capable of decoding

a.) MPEG1 and MPEG2 video, with DSS syntax extentions and constraints.
b.) MPEG2 MP@ML video
c.) MPEG2 MP@HL video with ATSC/GA constraints
d.) An MPEG2 aligned PES packet containing any of the above MPEG2 streams.

The scope of this section covers items "a", "b", and "c".

A3.2. Video Sequence

video_sequence() {	No. of bits	Mnemonic
next_start_code()		
sequence_header()		
if (nextbits() == extension_start_code) {		
sequence_extension()		
do {		
extension_and_user_data(0)		
do {		
if (next_bits() == group_start_code) {		
group_of_pictures_header()		
extension_and_user_data(1)		
}		
picture_header()		
extensions_and_user_data(2)		
picture_data()		
} while ((next_bits() == picture_start_code) ||		
next_bits() == group_start_code))		
if (nextbits() != sequence_end_code) {		
sequence_header()		
sequence_extension()		
}		
} while (nextbits() != sequence_end_code)		
} else {		

do {		
do {		
group_of_pictures_header()		
if (next_bits() == user_data_start_code)		
user_data()		
do {		
picture_header()		
if (next_bits() == user_data_start_code)		
user_data()		
picture_data()		
} while (next_bits() == picture_start_code)		
} while (next_bits() == group_start_code)		
if (nextbits() != sequence_end_code)		
sequence_header()		
} while (nextbits() != sequence_end_code)		
}		
sequence_end_code		
}		

A3.3. Sequence header

sequence_header() {	No. of bits	Mnemonic
sequence_header_code	32	bslbf
horizontal_size_value	12	uimsbf
vertical_size_value	12	uimsbf
pel_aspect_ratio	4	uimsbf
frame_rate	4	uimsbf
bit_rate	18	uimsbf
marker_bit	1	"1"
vbv_buffer_size	10	uimsbf
constrained_parameter_flag	1	
load_intra_quantizer_matrix	1	
if (load_intra_quantizer_matrix)		
intra_quantizer_matrix[64]	8*64	uimsbf
load_non_intra_quantizer_matrix	1	
if (load_non_intra_quantizer_matrix)		
non_intra_quantizer_matrix[64]	8*64	uimsbf
next_start_code()		

A3.4. Sequence extension

sequence_extension() {	No. of bits	Mnemonic
extension_start_code	32	bslbf
extension_start_code_identifier	4	uimsbf
profile_and_level_indication	8	uimsbf
non_interlaced_sequence	1	uimsbf
chroma_format	2	uimsbf
horizontal_size_extension	2	uimsbf
vertical_size_extension	2	uimsbf
bit_rate_extension	12	uimsbf
marker	1	
vbv_buffer_size_extension	8	uimsbf
frame_rate_extension	8	uimsbf
next_start_code()		
}		

A3.5. Extension and user data

extension_and_user_data(i) {	No. of bits	Mnemonic
while ((nextbits()==extension_start_code) ||		
(nextbits()==user_start_code)) {		
if (nextbits()==extension_start_code)		
extension_data(i)		
if (nextbits()==user_start_code)		
user_data()		
}		
}		

A3.6. User data

user_data() {	No. of bits	Mnemonic
user_data_start_code	32	bslbf
while(nextbits() != '0000 0000 0000 0000 0000 0001') {		
user_data	8	
}		
next_start_code()		
}		

A3.7. Sequence display extension

sequence_display_extension() {	No. of bits	Mnemonic
extension_start_code	32	bslbf
extension_start_code_identifier	4	uimsbf
video_format	3	uimsbf
colour_description	1	uimsbf
if (colour_description) {		
colour_primaries	8	uimsbf
transfer_characteristics	8	uimsbf
matrix_coefficients	8	uimsbf
}		
display_horizontal_dimension	14	uimsbf
marker_bit	1	"1"
display_vertical_dimension	14	uimsbf
next_start_code()		
}		

A3.8. Quant matrix

quant_matrix_extension() {	No. of bits	Mnemonic
extension_start_code	32	bslbf
extension_start_code_identifier	4	uimsbf
load_intra_quantizer_matrix	1	uimsbf
if (load_intra_quantizer_matrix)		
intra_quantizer_matrix[64]	8 * 64	uimsbf
load_non_intra_quantizer_matrix	1	uimsbf
if (load_non_intra_quantizer_matrix)		
non_intra_quantizer_matrix[64]	8 * 64	uimsbf
load_chroma_intra_quantizer_matrix	1	"0"
load_chroma_non_intra_quantizer_matrix	1	"0"
next_start_code()		
}		

A3.9. Group of pictures header

group_of_pictures_header() {	No. of bits	Mnemonic
group_start_code	32	bslbf
time_code	25	
closed_gop	1	
broken_link	1	
next_start_code()		
}		

A3.10. Picture header

picture_header() {	**No. of bits**	**Mnemonic**
picture_start_code	32	bslbf
temporal_reference	10	uimsbf
picture_coding_type	3	uimsbf
vbv_delay	16	uimsbf
if (picture_coding_type == 2 || picture_coding_type == 3) {		
full_pel_forward_vector	1	
forward_f_code	3	uimsbf
}		
if (picture_coding_type == 3) {		
full_pel_backward_vector	1	
backward_f_code	3	uimsbf
}		
while (nextbits() == '1') {		
extra_bit_picture	1	"1"
extra_information_picture	8	
}		
extra_bit_picture next_start_code()	1	"0"
}		

picture_coding_extension() {	No . of bits	Mnemonic
extension_start_code	32	bslbf
extension_id	4	uimsbf
forward_horizontal_f_code	4	uimsbf
forward_vertical_f_code	4	uimsbf
backward_horizontal_f_code	4	uimsbf
backward_vertical_f_code	4	uimsbf
intra_dc_precision	2	uimsbf
picture_structure	2	uimsbf
top_field_first	1	uimsbf
frame_pred_frame_dct	1	uimsbf
concealment motion vectors	1	uimsbf
q_scale_type	1	uimsbf
intra_vlc_format	1	uimsbf
alternate_scan	1	uimsbf
number_of_field_displayed_code	1	uimsbf
chroma_postprocessing_type	1	uimsbf
non_interlaced_frame	1	uimsbf
composite_display_flag	1	uimsbf
if (composite_display_flag) {		
v-axis	1	uimsbf
field_sequence	3	uimsbf
sub_carrier	1	
burst_amplitude	7	uimsbf
sub_carrier_phase	8	uimsbf
}		
next_start_code()		
}		

frame_pred_frame_dct is 1 indicates that the dct is frame based and the prediction is frames based and the prediction is 16x16 (as in MPEG-1). 0 enables all of the field dct, field pred and dual prime.

A3.11. Picture pan-scan extension

picture_pan_scan_extension() {	No. of bits	Mnemonic
extension_start_code	32	bslbf
extension_start_code_identifier	4	uimsbf
for (i=0; i<number_of_pan_offsets; i++) {		
pan_horizontal_left_upper_offset_integer	12	uimsbf
pan_horizontal_left_upper_offset_sub_pel	4	uimsbf
marker	1	
pan_vertical_left_upper_offset_integer	12	uimsbf
pan_vertical_left_upper_offset_sub_pel	4	uimsbf
marker	1	
}		
next_start_code()		
}		

```
if (non_interlaced_sequence)
        number_of_pan_offsets = 1
else
        number_of_pan_offsets = number_of_fields_displayed
```

A3.12. Picture Data

picture_data() {	No. of bits	Mnemonic
do {		
slice()		
} while (nextbits() == slice_start_code)		
next_start_code()		
}		

A3.13. Slice layer

slice() {	No. of bits	Mnemonic
slice_start_code	32	bslbf
quantizer_scale_code	5	uimsbf
while (nextbits() == '1') {		
extra_bit_slice	1	"1"
extra_information_slice	8	
}		
extra_bit_slice	1	"0"
do {		
macroblock()		
} while (nextbits() != '000 0000 0000 0000 0000 0000')		
next_start_code()		
}		

A3.14. Macroblock layer

macroblock() {	**No. of bits**	**Mnemonic**
if (<sequence extension was not present>)		
while (nextbits() == '0000 0001 111')		
macroblock_stuffing	11	vlclbf
while (nextbits() == '0000 0001 000')		
macroblock_escape	11	vlclbf
macroblock_address_increment	1-11	vlclbf
macroblock_type	1-8	vlclbf
if (macroblock_motion_forward ||		
macroblock_motion_backward) {		
if (picture_structure == 'frame') {		
if (frame_pred_frame_dct == 0)		
frame_motion_type	2	uimsbf
} else {		
field_motion_type	2	uimsbf
}		
}		
if ((picture_structure == 'frame') &&		
(frame_pred_frame_dct == 0) &&		
(macroblock_intra || macroblock_pattern))		
dct_type	1	uimsbf
if (macroblock_quant)		
quantizer_scale_code	5	uimsbf
if (macroblock_motion_forward ||		
(macroblock_intra && concealment_motion_vectors))		
forward_motion_vectors()	...	...
if (macroblock_motion_backward)		
backward_motion_vectors()	...	...
if (macroblock_intra && concealment_motion_vectors)		
marker_bit	1	
if (macroblock_pattern)		
coded_block_pattern()	...	...
for (i=0; i<block_count; i++) {		
block(i)		
}		
if (picture_coding_type == 4)		
end_of_macroblock	1	"1"
}		

motion_vectors () {	No. of bits	Mnemonic
if (motion_vector_count == 1) {		
if (mv_format == frame) {		
motion_vector()	...	...
} else {		
field_motion_vector()	...	...
}	...	...
} else {		
field_motion_vector()	...	...
field_motion_vector()	...	...
}		
}		

motion_vector () {	No. of bits	Mnemonic
motion_horizontal_code	1-13	vlclbf
if ((horizontal_f!=1) && (motion_horizontal_code != 0))		
motion_horizontal_r	1-8	uimsbf
if (dmv == 1)		
dmv_horizontal	1-2	vlcbf
motion_vertical_code	1-13	vlclbf
if ((vertical_f!=1) && (motion_vertical_code != 0))		
motion_vertical_r	1-8	uimsbf
if (dmv == 1)		
dmv_vertical	1-2	vlcbf
}		

field_motion_vector () {	No. of bits	Mnemonic
motion_vertical_field_select	1	uimsbf
motion_vector()	...	...
}		

coded_block_pattern () {	No. of bits	Mnemonic
coded_block_pattern_420	3-9	vlclbf
if ((chroma_format == 4:4:4) || (chroma_format == 4:2:2))		
extension of coded block pattern		
}		

block(i) {	No. of bits	Mnemonic
if (pattern_code[i]) {		
if (macroblock_intra) {		
if (i<4) {		
dct_dc_size_luminance	2-9	vlclbf
if(dct_dc_size_luminance != 0)		
dct_dc_differential	1-11	uimsbf
} else {		
dct_dc_size_chrominance	2-10	vlclbf
if(dct_dc_size_chrominance !=0)		
dct_dc_differential	1-11	uimsbf
}		
} else {		
First DCT coefficient	. . .	
}		
if (picture_coding_type != 4) {		
while (nextbits() != End of block)		
Subsequent DCT coefficients	. . .	
End of block	. . .	
}		
}		
}		

The start code prefix is "0000 0000 0000 0000 0000 0001

name	start code value (hexadecimal)
picture_start_code	00
slice_start_code	01 through AF
reserved	B0
reserved	B1
user_data_start_code	B2
sequence_header_code	B3
sequence_error_code	B4
extension_start_code	B5
reserved	B6
sequence_end_code	B7
group_start_code	B8
system start codes (see note)	B9 through FF

A3.15. Some Characteristics of MPEG2 Main Profile

1) Chroma Format: 4:2:0

2) Dual prime
Use only for M=1.

3) New alternate scan
A flag in the picture coding extension

4) Nonlinear Quantization scale
A flag in the picture coding extension

5) Quantization matrices
Download in the sequence and picture layer

6) Sequence header
Sequence header before any picture header is allowed

7) Intra VLC table
A flag in the picture coding extension

8) Escape coding
Double escape and 12 bit escape are both allowed. 12 bit escape is always used in MPEG 2 bitstreams

9) Error Concealment
Intra MV is allowed for error concealment. A flag in the picture coding extension

10) f-code
f-code is extended to 4 bits. Horizontal and vertical f codes are allowed

11) Motion vector search range in main profile
[127.5,128] with half-pel precision vertically. No restriction in horizontal direction

12) VBV buffer size
1.75 Mbits

13) Adaptive Frame/Field picture
Allowed in MPML

14) IDCT mismatch
New oddification proposal is always used in MPEG 2 bitstreams

15) MB stuffing
It is allowed for MPEG 1 compatibility. It is not allowed in MPEG 2 bitstream

16) DC precision is restricted to less than 10 bits in MLMP

A4. DSS VIDEO BITSTREAM CONSTRAINTS

A4.1. SCOPE

The HD-MPEG video decoder IC shall be capable of decoding

a.) MPEG1 and MPEG2 video, with DSS syntax extentions and constraints.
b.) MPEG2 MP@ML video
c.) MPEG2 MP@HL video with ATSC/GA constraints
d.) An MPEG2 aligned PES packet containing any of the above MPEG2 streams.

The scope of this section covers item "a".

The HD Video Decoder is expected to decode and display DSS streams. As such, this IC shall include all of the decoding and display functionality of the STi3500.

The Digital Satellite System (DSS) utilizes both ISO MPEG-1 Video (ISO/IEC 11172-2) as well as ISO MPEG-2 Mai r (MP) Main Level (ML) Video (ISO/IEC 13818-2) bit stream with some additional constraints. In this document, we note specific constraints of the DSS Video bit streams that are different from MPEG-1 and MPEG-2 MP, ML Video bit str ., also discuss the use of DSS user data in DSS Video.

Following list is a brief, non inclusive summary of the constraints in DSS video bit stream:

- Dual Prime is not used
- Concealment motion vectors are not used
- f_code=8 is not used
- Low Delay mode is not used
- D-pictures are not used
- Pan and Scan values below 1/4 pel resolution are not used
- Vertical Pan and Scan is not used
- Bit streams without fixed M and N structure and without B Frames at the beginning are not permitted (example structure that is not permitted: I P P P P P B B B B)

This document reflects the semantic constraints some of which are outlined in the above list.

A4.2. MPEG-1 Video Data Syntax in DSS

A4.2.1. DSS MPEG-1 Video Sequence

Sequence Header and Extensions:
zero stuff + **SH+** zero stuff +GOPH

SH: sequence_header()

There shall be a Sequence Header followed by a GOP Header for every I-frame in DSS Video bit stream. I.e., a Sequence Header will always precede a GOP Header.
Zero stuffing is allowed before sequence_header
No user_data() is allowed for DSS after sequence_header
Zero stuffing is allowed after Sequence Header

If a **sequence_end_code** is present, then sequence parameters can be changed in the next sequence layer. The **sequence_end_code** may be followed by zero stuffing, and shall be followed by the next Sequence Header.

Group_Of_Pictures_header:
zerostuff + SH +zerostuff+**GOPH**+PH

GOPH: group_of_pictures_header()

No extension or user_data allowed for DSS after group_of_pictures_header()
Zero stuffing is allowed before group_of_pictures_header()
No zero stuffing is allowed after group_of_pictures_header() , i.e., group_of_pictures_header() is immediately followed by picture_header() of the I Frame
(Note that zero stuffing between GOPH and Picture header may be introduced at the packetizer for packet alignment of Picture Header)
GOP headers never occur without a sequence header.

Picture Header:
PH+UD

PH: picture_header()
UD: user_data()

No zero stuffing is allowed after the above structure i.e.,

PH+UD+picture data

Each and every picture_header shall be followed by User_data.

The combined size of the GOP and Picture Headers (including the User data following the picture header) shall not exceed 122 Bytes.
i.e., **GOPH+PH+UD <123 Bytes**

A4.3. Video Sequence Header

The following fields (in italics) in the MPEG-1 Video Sequence header are constrained for DSS MPEG-1 Video:

Sequence Header

sequence_header() {	**No of bits**	**Mnemonic**
sequence_header_code	32	bslbf
horizontal_size	12	uimsbf
vertical_size	12	uimsbf
pel_aspect_ratio	4	uimsbf
picture_rate	4	uimsbf
bit_rate	18	uimsbf
marker_bit	1	"1"
vbv_buffer_size	10	uimsbf
constrained_parameter_flag	1	
load_intra_quantizer_matrix	1	
if (load_intra_quantizer_matrix)		
intra_quantizer_matrix[64]	8*64	uimsbf
load_non_intra_quantizer_matrix	1	
if (load_non_intra_quantizer_matrix)		
non_intra_quantizer_matrix[64]	8*64	uimsbf
next_start_code()		
}		

A4.3.1. Horizontal Size

Allowed horizontal sizes are: 720, 704, 544, 480, and 352

A4.3.2. Vertical Size

Allowed vertical sizes are: 480 and 240 for NTSC.

A4.3.3. Pel Aspect Ratio

Since some of the picture sizes defined in DSS are different from standard MPEG-1 defined picture sizes, the pel aspe that are used in the DSS are not present in the standard MPEG table (in section 2.4.3.2. of part 2 of ISO 11172 (MP Video)). Therefore, the only significance of the pel aspect ratio code in the DSS is to indicate whether the picture aspect ra 4:3 or 16:9.

- For picture aspect ratio of 4:3, the pel aspect ratio should be set to 1100b
- For picture aspect ratio of 16:9, the pel aspect ratio should be set to 0110b

The supported frame sizes and pel aspect ratios are tabulated below:

Resolution	Aspect Ratio	Video Standard	Horiz. Size	Vert. Size	Pel Aspect Ratio	
					value	code
CCIR-601	4:3	NTSC	720 (704*)	480	1.1000	1100
CCIR-601	16:9	NTSC	720 (704*)	480	0.8250	0110
Large	4:3	NTSC	544	480	0.8500	1100
Large	16:9	NTSC	544	480	0.6375	0110
Medium	4:3	NTSC	480	480	0.7500	1100
Medium	16:9	NTSC	480	480	0.5625	0110
Small	4:3	NTSC	352	480	0.5500	1100
Small	16:9	NTSC	352	480	0.4125	0110
SIF	4:3	NTSC	352	240	1.1000	1100

* DSS decoders can handle 720 as well as 704 horizontal size.

A4.3.4. Picture Rate

In the DSS, this field indicates the display rate, and hence it is always set to 0100b (NTSC, 29.97 frames/sec). In the case of film mode (3:2 pull down), the frame rate code is set to 0100b, and 3:2 pull down information is conveyed thr field display flags in the picture user data.

A4.3.5. Bit Rate Value

The maximum allowed video bit rate is $15x10^6$ bits/s. For Statistical Multiplexing applications, this field is set to all "1"s: i.e 3FFFF.

A4.3.6.VBV Buffer Size Value

The encoder VBV Buffer size is constrained to be equal or less than 1,835,008 bits (vbv_buffer_size=112) for Constant Rate operation.
The Encoders have to make sure that the decoder VBV buffers never under- or over-flow.

Note: Current implementation of Statistical Multiplexing based Variable bit rate operation uses 1,200,000 bits for encoder buffer and 3,532,800 bits for decoder VBV buffer. The goal is to reduce the decoder VBV size to 2,900,000 bits by the e second Quarter 95, and eventually to 1,835,008 bits.

A4.3.7. Constrained Parameters Flag

In DSS this bit is always set to 0.

A4.4. Group of Pictures Header

No extension or user_data allowed for DSS after group_of_pictures_header().
No zero stuffing is allowed after group_of_pictures_header() , i.e., group_of_pictures_header() is immediately followed by picture_header() of the I Frame
(Note that zero stuffing between GOPH and Picture header may be introduced at the packetizer for packet alignment of Picture Header)
GOP headers never occur without a sequence header.

A4.5. Picture Header

The following fields (in italics) in the MPEG-1 Video picture header are constrained for DSS MPEG-1 Video:

Picture header

```
picture() {

  picture_start_code                                            32        bslbf
  temporal_reference                                            10        uimsbf
  picture_coding_type                                           3         uimsbf
  vbv_delay                                                     16        uimsbf
    if ( picture_coding_type == 2 || picture_coding_type == 3) {
      full_pel_forward_vector                                   1
      forward_f_code                                            3         uimsbf
    }
    if ( picture_coding_type == 3 ) {
      full_pel_backward_vector                                  1
      backward_f_code                                           3         uimsbf
      }
    while ( nextbits() == '1' ) {
      extra_bit_picture                                         1         "1"
      extra_information_picture                                 8
    }
    extra_bit_picture                                           1         "0"
    next_start_code()
  }
```

A4.5.1. Picture Coding Type

In the DSS, no D pictures are allowed (code 100b is not allowed)

A4.5.2. Extra Bit Picture

In the DSS, this bit is always set to 0.

A4.6. Video User Data

DSS utilizes the user data field following the picture header to convey picture related information such as presentation decode time stamps for audio-video synchronization, chroma post processing information, pan and scan information for dis of 16:9 images on 4:3 TV sets, 2:3 pull down information for efficient coding and display of movie originated materia frames/s) in 59.94 fields/s NTSC system, color burst suppression information, closed caption and extended data services. Note that DSS does not support the use of user data in sequence header or GOP header levels. By conveying the user information in picture header level, DSS achieves automatic synchronization of picture related data with the picture t belongs to.

User data field is allowed only following the picture header. Each and every picture header is followed by user data. Only user data field (with possibly multiple user_data_types) is allowed per picture. The User_data syntax is extensible for f applications. Note that there is an escape mechanism to extend the number of "type" numbers, using user_data_t Secondly, each User_data record is an integer number of bytes.

user_data() {	**No. of bits**	**Mnemonic**
user_data_start_code	32	bslbf
while(nextbits() != '0000 0000 0000 0000 0000 0001') {		
user_data_length	8	uimsbf
user_data_type	8	uimsbf
if (user_data_type==0xFF)		
ext_user_data_type	8	uimsbf
user_data_info()	(user_data_length-1)*8	uimsbf
}		
next_start_code()		
}		

A4.6.1. User Data Start Code

This field is set to MPEG user_data_start_code = 0x 000001B2

A4.6.2. User Data Length

This field indicates the length in bytes of user_data_type and user_data_info fields.
If the user_data_type is not recognized, the decoder will skip over user_data_length-1 bytes to find the next user_data_length field.

A4.6.3. User Data Type

The following user data types are defined for DSS:

8-bit code	Type	user_data_length
0x00	forbidden	-
0x01	reserved	-
0x02	presentation_time_stamp	1+5
0x03	reserved	-
0x04	decode_time_stamp	1+5
0x05	chroma_flags	1+1
0x06	pan_and_scan	1+2
0x07	fields_display_flags	1+1
0x08	no_burst	1+0
0x09	closed_caption	1+2
0x0A	extended_data_services	1+2
0x0B-0xFE	reserved	-
0xFF	escape to ext_user_data_type	1+0

A4.6.4. Ext User Data Type

Escape mechanism to further define 255 user data types in future (reserved, do not use)

A4.6.5. User Data Info

user_data_info() {	**No. of bits**	**Mnemonic**
switch (**user_data_type**){		
case **presentation_time_stamp**:		
six_bit_pad	6	"000000"
presentation_time_stamp[31 ..30]	2	bslbf
marker_bit	1	"1"
presentation_time_stamp[29..15]	15	bslbf
marker_bit	1	"1"
presentation_time_stamp[14..0]	15	bslbf
break		

case **decode_time_stamp**:		
six_bit_pad	6	"000000"
decode_time_stamp[31 ..30]	2	bslbf
marker_bit	1	"1"
decode_time_stamp[29..15]	15	bslbf
marker_bit	1	"1"
decode_time_stamp[14..0]	15	bslbf
break		

case **chroma_flags**:		
frame_filter	1	uimsbf
horizontal_sampled_422	1	uimsbf
vertical_sampled_422	1	uimsbf
chroma_spare_flag_1	1	uimsbf
chroma_spare_flag_2	1	uimsbf
chroma_spare_flag_3	1	uimsbf
chroma_spare_flag_4	1	uimsbf
chroma_spare_flag_5	1	uimsbf
break		

case **pan_and_scan**:		
pan_and_scan	12	simsbf
marker_bit	1	"1"
three_bit_pad	3	"000"
break		

case **fields_display_flags**:		
field_display_parity	1	uimsbf
field_count	3	uimsbf
field_spare_1	1	uimsbf
field_spare_2	1	uimsbf
field_spare_3	1	uimsbf
field_spare_4	1	uimsbf
break		

case **no_burst**:		
/* No data is associated with no_burst */		
break		

case **closed_caption:**		
closed_caption_byte1	8	uimsbf
closed_caption_byte2	8	uimsbf
break		

case **extended_data_services:**		
extended_data_services_byte1	8	uimsbf
extended_data_services_byte2	8	uimsbf
break		

default:		
skip_bytes(user_data_length-1)		
}		

A4.6.5.1. Presentation Time Stamp

The PTS is a 32-bit number coded in three separate fields. It indicates the intended time of presentation in the decoder of the associated frame. The value of PTS is measured in the number of periods of the 27 MHz system clock.
PTS always has to be the first user_data_info in user data field.

A4.6.5.2. Decode Time Stamp

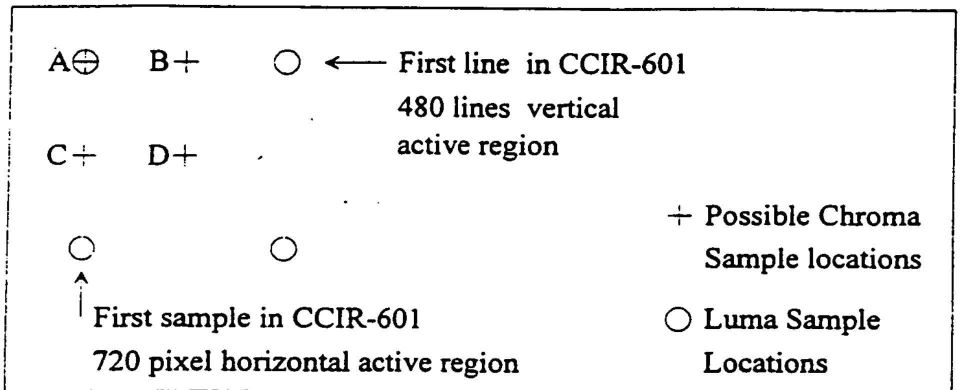
The DTS is a 32-bit number coded in three separate fields. It indicates the intended time of decoding of the associated frame in the decoder. The value of DTS is measured in the number of periods of the 27 MHz system clock.
If DTS field is not present, the decoders assume a GOP structure with M=3 (2 B frames between I or P frames).

A4.6.5.3. Chroma Flags

frame_filter: This flag indicates if the encoder prefiltered the chroma on a field or frame basis prior to subsampling vertically. A value of "1" indicates frame processing. Value "0" indicates that each chroma field was independently filtered and sub sampled vertically.
The last value of the frame_filter is used until a different one is received. If no chroma flag user data is received in the bit stream, then the default value is 1, i.e., frame processing.
Default value is used after each bit stream switch until a new value is received.

horizontal_sampled_422: This flag indicates the horizontal location of the chroma sample within the associated quad of luma samples with respect to upper-left luma sample: A value of "1" indicates horizontally co-located. Value "0" indicates horizontally halfway. See Figure A2 for illustration.
The last value of the horizontal_sampled_422 flag is used until a different one is received. If no chroma flag user data received in the bit stream, then the default value is 1, i.e., co-located.
Default value is used after each bit stream switch until a new value is received.

vertical_sampled_422: This flag indicates the vertical location of the chroma sample within the associated quad of luma samples with respect to upper-left luma sample: A value of "1" indicates vertically co-located. Value "0" indicates vertically halfway. See Figure A2 for illustration.
The last value of the vertical_sampled_422 flag is used until a different one is received. If no chroma flag user data is received in the bit stream, then the default value is 0, i.e., halfway.
Default value is used after each bit stream switch until a new value is received.



	A	B	C	D
horizontal_sampled_422	1	0	1	0
vertical_sampled_422	1	1	0	0

Figure A2

chroma_spare_flag_1 reserved
chroma_spare_flag_2 reserved
chroma_spare_flag_3 reserved
chroma_spare_flag_4 reserved
chroma_spare_flag_5 reserved

A4.6.5.4. Pan and Scan

Pan_and_Scan value indicates the horizontal offset for display of 16:9 material on a 4:3 monitor. By default, the ima centered in the frame. The pan_and_scan information gives an offset to quarter pixel resolution of the desired left edge rel to the default left edge. The encoder will scale the Pan_and_Scan values for resolution dependence for sub sampled for Hence, the Pan_and_Scan information in the user data will be in terms of sub sampled format.

DSS user data

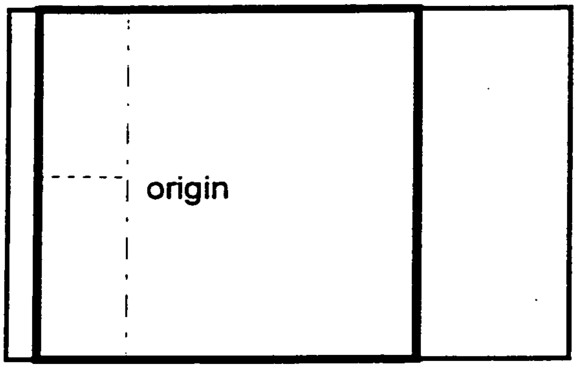


Figure A3

- For a picture, if there is no pan_and_scan information present in the user data field, the default value is 0 (ima centered in the frame)

- Within a picture, if there is only one pan_and_scan information, this value is valid for all fields to be displayed from picture. e. g.,

  3 fields to be displayed from a particular picture with the order, f1 f2 f1
  1 pan_and_scan information is present: p1
  the pan_and_scan information used for each field would be:

f1	f2	f1
p1	p1	p1

- Within a picture, if there are as many pan_and_scan information as there are fields to be displayed from that pic the pan_and_scan information is used in order. e. g.,

  3 fields to be displayed from a particular picture with the order, f1 f2 f1
  3 pan_and_scan information is present: p1 p2 p3
  the pan_and_scan information used for each field would be:

f1	f2	f1
p1	p2	p3

- Within a picture, if the number of pan_and_scan information is less than the number of fields to be displayed from picture, the pan_and_scan information is used in order for each displayed field, and the last pan_and_scan information is for all the remaining fields of that picture. e. g.,

  3 fields to be displayed from a particular picture with the order, f1 f2 f1
  2 pan_and_scan information is present: p1 p2
  the pan_and_scan information used for each field would be:

f1	f2	f1
p1	p2	p2

Pan and Scan information in the MPEG 1 DSS bit streams will be conveyed as:

Sequence Header
pel_aspect_ratio = 0x06

Picture Header
Picture User Data
DSS's pan_and_scan user data

Decoder actions for the MPEG 1 case

P&S in userdata	Decoder's action
Yes	Use the transmitted one
No	Set pan_and_scan = 0

A4.6.5.5. Field Display Flags
Field_Display_Flags instruct the display device how to display the present frame of video. The main use is for 3.2 pull-d where irregular editing of pulled-down material is encountered. The two codes are the following:

first_field_parity: This 1-bit-flag is used to identify the parity of the first displayed field,

- '0' indicates to display the **top field** of the frame as the first field
- '1' indicates to display the **bottom field** of the frame as the first field

field_count: This three bit field indicates the number of fields to be displayed from this frame. The actual number of fi displayed is field_count+1. The field_count can be between 1 and 7, indicating the case of 2....., 8 fields being displayed the single received frame. The parity of the displayed field alternates with each displayed field (example: top, bottom, top,...)

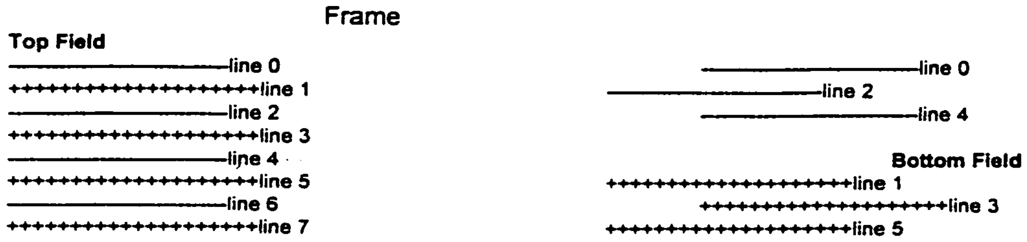


Figure A4

A4.6.5.6. No Burst

The broadcaster may require that if the original material is Black & White (no-color burst on the original video tape), the o should not contain color burst. This requirement can be met with this flag. The occurrence of this No_Burst in User_dat result in the display processor suppressing burst on its output. The mere occurrence of the "No_Burst" type is suffi information for the decoder. If no No_Burst flag is received, the color burst is generated.

A4.6.5.7. Closed Caption

Closed captions in NTSC are conveyed by inserting two ASCII characters on scan-line 21 in the VBI of Field 1. In DSS, it wi supported by an encoder circuit which strips these bytes for transmission as video User_Data. The ASCII data is stripp the encoder and transmitted as 8-bit character data. The receiver inserts the closed_caption by reconstructing scan_lin with its sine-wave sync pulse and NRZ bit wave form. Since closed caption data from two "Field 1's" may have t transmitted in a single frame (a possible consequence of removing repeated fields in material that went through 3:2 pull do the Closed_caption User_Data may occur twice in a single Picture. In this case it should be used in order of reception.

A4.6.5.8. Extended Data Services

The DSS User_Data supports insertion of two ASCII characters on line 21 Field 2 in an identical fashion to Closed_caption.

A4.6.6. General Information about Video_user_data in DSS MPEG-1 Video

As noted before, video user data field is only permitted in picture header level (not in Sequence header or GOP header). a user_data_field is opened, there may be different types of user_data present in the user_data_field. Some of the user_ information, such as pan_and_scan, closed_captioning, and extended_data_services, may be related to the number c displayed from one frame. Therefore, in case that more than two fields are displayed from the same frame, some user_data types may occur more than once in the same user_data_field.

The following table summarizes these cases

User data type	max allowed in one user_data_field	Meaning
presentation_time_stamp	1	maximum 1 PTS for each frame
decode_time_stamp	1	maximum 1 DTS for each frame
chroma_flags	1	
pan_and_scan	< = number of fields to be displayed	Each field may have different pan&scan

fields_display_flags	1	
no_burst	1	
closed_caption	< = number of odd fields to be displayed	each odd field can carry 2 characters
extended_data_services	< = number of even fields to be displayed	each even field can carry 2 characters

A5. MPEG-2 VIDEO DATA SYNTAX IN DSS

A5.1. SCOPE

The HD-MPEG video decoder IC shall be capable of decoding

a.) MPEG1 and MPEG2 video, with DSS syntax extentions and constraints.
b.) MPEG2 MP@ML video
c.) MPEG2 MP@HL video with ATSC/GA constraints
d.) An MPEG2 aligned PES packet containing any of the above MPEG2 streams.

The scope of this section covers item "a".

The HD Video Decoder is expected to decode and display DSS streams. As such, this IC shall include all of the decoding and display functionality of the STi3500.

A5.2. Video Sequence Header

Sequence Header and Extensions: zerostuff + **SH** + **SE** + **SDE** + zerostuff

SH: sequence_header()
SE: sequence_extension()
SDE: (optional) sequence_display_extension()

There shall be a Sequence Header followed by a GOP Header for every I-frame in DSS Video bit stream.
Zero stuffing is allowed before sequence_header
No user_data() is allowed for DSS after sequence_header
Zero stuffing is allowed after sequence_display_extension()

If a **sequence_end_code** is present, then sequence parameters can be changed in the next sequence layer. The **sequence_end_code** may be followed by zero stuffing, and shall be followed by the next Sequence Header.

Group_Of_Pictures_header: zerostuff + SH + SE + SDE + zerostuff+**GOPH**+**PH**

GOPH: group_of_pictures_header()

No extension or user_data allowed for DSS after group_of_pictures_header()
Zero stuffing is allowed before group_of_pictures_header()
No zero stuffing is allowed after group_of_pictures_header() , i.e., group_of_pictures_header() is always preceded by sequence_header and extensions and is immediately followed by picture_header() of the I Frame
(Note that zero stuffing between GOPH and Picture header may be introduced at the packetizer for packet alignment of Picture Header)
GOP headers never occur without a sequence header.

Picture Header: **PH+PCE+PDE+UD+QME**

PH: picture_header()
PCE: picture_coding_extension()
PDE: (optional) picture_display_extension()
UD: user_data()

QME: (optional)Quant_matrix_extension()

Each and every picture_header shall be followed by User_data.

Quant_matrix_extension() if exists, has to follow user_data()
No zero stuffing is allowed after the above structure, i.e., PH+PCE+PDE+UD+QME+pict. data

The combined size of the GOP and Picture Headers (including the picture coding extension, picture display extension and User data, but excluding the optional Quant matrix extension) shall not exceed 122 Bytes.
i.e., **GOPH+PH+PCE+PDE+UD <123 Bytes**

A5.3. Video Sequence Header

The following fields (in italics) in the MPEG-2 Video Sequence header are further constrained (in addition to Main Profile Level constraints) for DSS MPEG-2 Video :

Sequence Header

sequence_header() {	**No. of bits**	**Mnemonic**
sequence_header_code	32	bslbf
horizontal_size_value	12	uimsbf
vertical_size_value	12	uimsbf
aspect_ratio_information	4	uimsbf
frame_rate_code	4	uimsbf
bit_rate_value	18	uimsbf
marker_bit	1	"1"
vbv_buffer_size_value	10	uimsbf
constrained_parameters_flag	1	
load_intra_quantiser_matrix	1	
if (load_intra_quantiser_matrix)		
intra_quantiser_matrix[64]	8*64	uimsbf
load_non_intra_quantiser_matrix	1	
if (load_non_intra_quantiser_matrix)		
non_intra_quantiser_matrix[64]	8*64	uimsbf
next_start_code()		
}		

A5.3.1. Horizontal Size Value

Allowed horizontal sizes are: 720, 704, 544, 480, and 352

A5.3.2. Vertical Size Value

Allowed vertical sizes are: 480, and 240 for NTSC.

A5.3.3. Aspect Ratio Information

The only allowed aspect ratio information codes are:
0010b For picture aspect ratio of 4:3
0011b For picture aspect ratio of 16:9

The allowed picture formats and aspect ratio information are:

Resolution	Aspect Ratio	Video Standard	Horiz. Size	Vert. Size	Aspect Ratio Information code
CCIR-601	4:3	NTSC	720 (704*)	480	0010
CCIR-601	16:9	NTSC	720 (704*)	480	0011
Large	4:3	NTSC	544	480	0010
Large	16:9	NTSC	544	480	0011
Medium	4:3	NTSC	480	480	0010
Medium	16:9	NTSC	480	480	0011
Small	4:3	NTSC	352	480	0010
Small	16:9	NTSC	352	480	0011
SIF	4:3	NTSC	352	240	0010

A5.3.4. Frame Rate Code

In the DSS, this field indicates the display rate, and hence it is always set to 0100b (NTSC, 29.97 frames/sec).
In the case of film mode (3:2 pull down), the frame rate code is set to 0100b, and 3:2 pull down information is conveye
1)through field display flags in the picture user data or 2)picture coding extension.

A5.3.5. Bit-Rate Value

The maximum allowed video bit rate is 15x10^6 bits/s. For Statistical Multiplexing applications, this field is set to all "1"s: i.e 3FFFF.

A5.3.6. VBV Buffer Size Value

The encoder VBV Buffer size is constrained to be equal or less than 1,835,008 bits (vbv_buffer_size=112) for Constant Rate operation.
The Encoders have to make sure that the decoder VBV buffers never under- or over-flow.

Note: Current implementation of Statistical Multiplexing based Variable bit rate operation uses 1,200,000 bits for encoder buffer and 3,532,800 bits for decoder VBV buffer. The goal is to reduce the decoder VBV size to 2,900,000 bits by the e second Quarter 95, and eventually to 1,835,008 bits.

A5.3.7. Constrained Parameters Flag

In DSS this bit is always set to 0

A5.4. Video Sequence Extension

The following fields (in italics) in the MPEG-2 Video Sequence extension are further constrained (in addition to Main Pr Main Level constraints) for DSS MPEG-2 Video :

Sequence extension

sequence_extension() {	No. of bits	Mnemonic
extension_start_code	32	.bslbf
extension_start_code_identifier	4	uimsbf
profile_and_level_indication	8	uimsbf
progressive_sequence	1	uimsbf
chroma_format	2	uimsbf
horizontal_size_extension	2	uimsbf
vertical_size_extension	2	uimsbf
bit_rate_extension	12	uimsbf
marker_bit	1	"1"
vbv_buffer_size_extension	8	uimsbf
low_delay	1	uimsbf
frame_rate_extension_n	2	uimsbf
frame_rate_extension_d	5	uimsbf
next_start_code()		
}		

A5.4.1. Progressive Sequence

This bit is always set to "0".

A5.4.2. Chroma Format

This field is always set to 01b indicating 4:2:0 chrominance format.

A5.4.3. Horizontal Size Extension

This field is always set to 0.

A5.4.4. Vertical Size Extension

This field is always set to 0.

A5.4.5. Bit Rate Extension

This field is always set to 0.

A5.4.6. VBV Buffer Size Extension

This field is always set to 0.

A5.4.7. Low Delay

This flag is always set to 0.

A5.5. Video Sequence Display Extension

The following fields (in italics) in the MPEG-2 Video Sequence display extension are further constrained (in addition to Profile Main Level constraints) for DSS MPEG-2 Video:

Sequence display extension

sequence_display_extension() {	No. of bits	Mnemonic
extension_start_code_identifier	4	uimsbf
video_format	3	uimsbf
colour_description	1	uimsbf
if (colour_description) {		
colour_primaries	8	uimsbf
transfer_characteristics	8	uimsbf
matrix_coefficients	8	uimsbf
}		
display_horizontal_size	14	uimsbf
marker_bit	1	"1"
display_vertical_size	14	uimsbf
next_start_code()		
}		

A5.5.1. Video Format

This field is always set to 010b to indicate NTSC.

A5.5.2. Color Description

This flag is always set to 0.

A5.5.3. Display Horizontal Size

This field has always the value 711 decimal. Combined with picture display extension field, this field is used to convey pan and scan information.

A5.5.4. Display Vertical Size

This field has always the value 483 decimal. Combined with picture display extension field, this field is used to convey pan and scan information.

A5.6. Group of Pictures Header

No extension or user_data allowed for DSS after group_of_pictures_header().

No zero stuffing is allowed after group_of_pictures_header() , i.e., group_of_pictures_header() is immediately followed by picture_header() of the I Frame.

(Note that zero stuffing between GOPH and Picture header may be introduced at the packetizer for packet alignment of Picture Header)

GOP headers never occur without a sequence header.

A5.7. Picture Header

The following fields (in italics) in the MPEG-2 Video picture header are further constrained (in addition to Main Profile Level constraints) for DSS MPEG-2 Video:

Picture header

picture_header() {	No. of bits	Mnemonic
picture_start_code	32	bslbf
temporal_reference	10	uimsbf
picture_coding_type	3	uimsbf
vbv_delay	16	uimsbf
if (picture_coding_type == 2 || picture_coding_type == 3) {		
full_pel_forward_vector	1	
forward_f_code	3	uimsbf
}		
if (picture_coding_type == 3) {		
full_pel_backward_vector	1	
backward_f_code	3	uimsbf
}		
while (nextbits() == '1') {		
extra_bit_picture	1	"1"
extra_information_picture	8	
}		
extra_bit_picture	1	"0"
next_start_code()		
}		

A5.7.1. Picture Coding Type

In the DSS, no D pictures are allowed (code 100b is not allowed in MPEG-2)

A5.7.2. Extra Bit Picture

In the DSS, this bit is always set to 0.

A5.8. Picture Coding Extension

The following fields (in italics) in the MPEG-2 Video picture coding extension are further constrained (in addition to Main Pr Main Level constraints) for DSS MPEG-2 Video:

Picture coding extension

picture_coding_extension() {	No . of bits	Mnemonic
extension_start_code	32	bslbf
extension_start_code_identifier	4	uimsbf
f_code[0][0] /* forward horizontal */	4	uimsbf
f_code[0][1] /* forward vertical */	4	uimsbf
f_code[1][0] /* backward_horizontal */	4	uimsbf
f_code[1][1] /* backward_vertical */	4	uimsbf
intra_dc_precision	2	uimsbf
picture_structure	2	uimsbf
top_field_first	1	uimsbf
frame_pred_frame_dct	1	uimsbf
concealment_motion_vectors	1	uimsbf
q_scale_type	1	uimsbf
intra_vlc_format	1	uimsbf
alternate_scan	1	uimsbf
repeat_first_field	1	uimsbf
chroma_420_type	1	uimsbf
progressive_frame	1	uimsbf
composite_display_flag	1	uimsbf
if (composite_display_flag) {		
v_axis	1	uimsbf
field_sequence	3	uimsbf
sub_carrier	1	
burst_amplitude	7	uimsbf
sub_carrier_phase	8	uimsbf
}		
next_start_code()		
}		

A5.8.1. F_code[0][0]

DSS only supports f_code[0][0] values 1-7 (f_code[0][0] value 8 not allowed).
If f_code is not used, it shall take the value 15 (all ones).

A5.8.2. F_code[1][0]

DSS only supports f_code[1][0] values 1-7 (f_code[1][0] value 8 not allowed)
If f_code is not used, it shall take the value 15 (all ones).

A5.8.3. Picture Structure

This field is always set to 11b (frame pictures). DSS does not support field pictures.

A5.8.4. Concealment Motion Vectors

This flag is set to "0". DSS does not support concealment motion vectors.

A5.8.5. Chroma 420 Type

If this bit is set to "0": Field chroma post processing
If this bit is set to "1": Frame chroma post processing

A5.8.6. Burst Amplitude

This 7 bit integer defines the burst amplitude for NTSC.

Refer to Section 3.9.7.4 for description of how to use this field in DSS.

A5.9. Picture Display Extension

The following fields (in italics) in the MPEG-2 Video picture display extension are further constrained (in addition to Main Pr Main Level constraints) for DSS MPEG-2 Video :

Picture display extension

picture_display_extension() {	No. of bits	Mnemonic
extension_start_code_identifier	4	uimsbf.
for (i=0; i<number_of_frame_centre_offsets; i++) {		
frame_centre_horizontal_offset	16	simsbf
marker_bit	1	"1"
frame_centre_vertical_offset	16	simsbf
marker_bit	1	"1"
}		
next_start_code()		
}		

A5.9.1. Frame Center Horizontal Offset

DSS does not support resolution higher than 1/4 pixel.

Refer to Section 3.9.7.2 for description of how to use this field in DSS.

A5.9.2. Frame Center Vertical Offset

DSS does not utilize vertical pan and scan. This field is always set to 0.

Refer to Section 3.9.7.2 for description of how to use this field in DSS.

A5.10. Video User Data in DSS MPEG-2 Video

DSS utilizes the user data field following the picture header to convey picture related information such as presentation decode time stamps for audio-video synchronization, chroma post processing information, pan and scan information for dis of 16:9 images on 4:3 TV sets, 2:3 pull down information for efficient coding and display of movie originated materia frames/s) in 59.94 fields/s NTSC system, color burst suppression information, closed caption and extended data services. Note that DSS does not support the use of user data in sequence header or GOP header levels. By conveying the user information in picture header level, DSS achieves automatic synchronization of picture related data with the picture t belongs to.

User data field is allowed only following the picture header. Only one user data field (with possibly multiple user_data_type allowed per picture. The User_data syntax is extensible for future applications. Note that there is an escape mechanis extend the number of "type" numbers, using user_data_type. Secondly, each User_data record is an integer number of byt

user_data() {	No. of bits	Mnemonic
user_data_start_code	32	bslbf
while(nextbits() != '0000 0000 0000 0000 0000 0001') {		
user_data_length	8	uimsbf
user_data_type	8	uimsbf
if (user_data_type==0xFF)		
ext_user_data_type	8	uimsbf
user_data_info()	(user_data_length-1)*8	uimsbf
}		
next_start_code()		
}		

A5.10.1. User Data Start Code

This field is set to MPEG user_data_start_code = 0x 000001B2

A5.10.2. User Data Length

This field indicates the length in bytes of user_data_type and user_data_info fields.
If the user_data_type is not recognized, the decoder will skip over user_data_length-1 bytes to find the next user_data_length field.

A5.10.3. User Data Type

The following user data types are defined for DSS:

8-bit code	Type	user_data_length
0x00	forbidden	-
0x01	reserved	-
0x02	presentation_time_stamp	1+5
0x03	reserved	-
0x04	decode_time_stamp	1+5
0x05	chroma_flags	1+1
0x06	pan_and_scan	1+2
0x07	fields_display_flags	1+1
0x08	no_burst	1+0
0x09	closed_caption	1+2
0x0A	extended_data_services	1+2
0x0B-0xFE	reserved	-
0xFF	escape to ext_user_data_type	1+0

A5.10.4. Ext User Data Type

Escape mechanism to further define 255 user data types in future (reserved, do not use)

A5.10.5. User Data Info

The Syntax and Semantics of the User Data info field in the picture user data field of MPEG-2 Video bit stream is identical to the User Data info field syntax and semantics explained in Section 2.5.5.

A5.10.6. General Information about Video_user_data in DSS MPEG-2 Video

As noted before, video user data field is only permitted in picture header level (not in Sequence header or GOP header). a user_data_field is opened, there may be different types of user_data present in the user_data_field. Some of the use information, such as pan_and_scan, closed_captioning, and extended_data_services, may be related to the number of displayed from one frame. Therefore, in case that more than two fields are displayed from the same frame, some user_data types may occur more than once in the same user_data_field.

The following table summarizes these cases

User data type	max allowed in one user_data_field	Meaning
presentation_time_stamp	1	maximum 1 PTS for each frame
decode_time_stamp	1	maximum 1 DTS for each frame
chroma_flags	1	
pan_and_scan	<= number of fields to be displayed	Each field may have different pan&scan

fields_display_flags	1	
no_burst	1	
closed_caption	< = number of odd fields to be displayed	each odd field can carry 2 characters
extended_data_services	< = number of even fields to be displayed	each even field can carry 2 characters

A5.10.7. Video User Data and MPEG-2 Fields

MPEG 2 Syntax supports some of the functions that are part of the DSS Video User Data, such as Chroma Flags, Pan and Scan, Field Display Flags and No Burst. In this section, we compare the DSS's user data and the MPEG 2 syntax, and summarizes the action that the decoder takes.

A5.10.7.1. Chroma_Flags and MPEG-2 Chroma_420_type Field

MPEG 2	**DSS's user data**
chroma_420_type	**chroma_flags**
1 bit	8 bit
	frame_filter 1 bit horizontal_pos 1 bit vertical_pos 1 bit spare_flags 5 bits

chroma_420_type = 0 => field chroma postprocessing
chroma_420_type = 1 => frame chroma postprocessing
	frame_filter = 1 => frame processing otherwise field processing
	horizontal_pos = 1 => collocated otherwise halfway
	vertical_pos = 1 => collocated otherwise halfway

Decoder actions:

- If chroma_flag is not in user data, the decoder follows the MPEG 2 interpretation of chroma_420_type
- If chroma_flag is in user data, the decoder follows the DSS's user data semantics.

A5.10.7.2. Pan and Scan and MPEG-2 Picture Display Extension

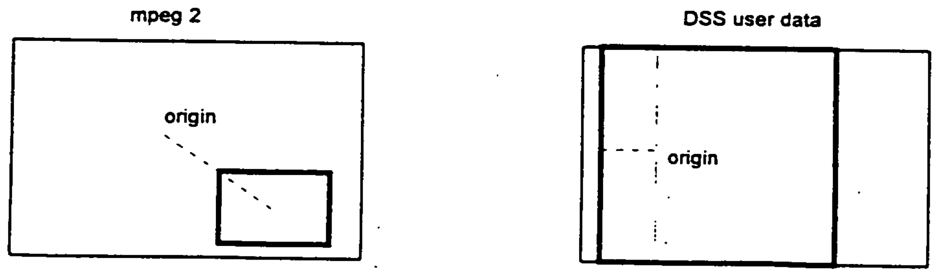


Figure A5

MPEG 2 bitstreams with pan_and_scan information using MPEG-2 fields should be:

Sequence Header aspect_ratio_information 0x3
Sequence Extension
Sequence Display Extension display_horizontal_size 711
display_vertical_size
483

Picture Header
Picture Coding Extension
Picture Display Extension MPEG 2's pan_and_scan

or using the pan_and_scan information in DSS picture user data

Sequence Header aspect_ratio_information 0x3
Sequence Extension
Sequence Display Extension (optional) display_horizontal_size 711

display_vertical_size 483

Picture Header
Picture Coding Extension
Picture User Data DSS's pan_and_scan user data

Decoder actions

Picture display extension in MPEG 2	Pan_and_scan in DSS User data	Decoder actions
Yes	Yes	Uses the P&S in user data and follows user data semantics
No	No	Reuses the last transmitted P&S until next Sequence Header. At next sequence header, P&S is reset to 0.
Yes	No	Uses the P&S in MPEG2 and follows MPEG2 semantics
No	Yes	Uses the P&S in user data and follows user data semantics

A5.10.7.3. Fields Display Flags and MPEG-2 Picture Coding Extension

	MPEG 2	DSS's user data (field_display_flags)
Name	**top_field_first**	**first_field_parity**
	1 bit	1 bit
	top_field_first =1 top field first	first_field_parity=0 top field first
Name	**repeat_first_field**	**field_count**
	1 bit	3 bits
	repeat_first_field =1 field displayed in 3 fields time	fields are displayed in field_count+1 time

Decoder actions

a) When the field_display_flags are not in the user data, the decoder performs like a normal MPEG2 decoder, and obeys all the semantics to interpret the top_field_first and repeat_first_field.

b) When the field_display_flags are in the user data, the decoder ignores the top_field_first and repeat_first_field, and obeys the semantics of DSS's user data.

A5.10.7.4. No Burst Flag and MPEG-2 Picture Coding Extension

MPEG 2	DSS's user data
burst_amplitude	**no burst flag**
7 bits	only user data type (no additional bit)

Decoder actions

a) When no_burst_flag is in user data, the decoder sets to no burst, and ignore the burst_amplitude.

b) If the no_burst_flag is not in user data, the decoder will check the burst_amplitude (if it is transmitted), and if burst_amplitude = 0, then the decoder sets to no burst. If the burst_amplitude is not zero, the decoder will ignore it.

A6. REFERENCES

A6.1. Normative References

The following documents contain provisions which, through reference in this text, constitute provisions of this standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreement based on this standard are encouraged to investigate the possibility of applying the most recent editions of the documents listed below.

ISO/IEC IS 13818-1, International Standard (1994), *MPEG-2 Systems.*

ISO/IEC IS 13818-2, International Standard (1994), *MPEG-2 Video.*

A6.2. Informative references

SMPTE 274M (1995), *Standard for television, 1920 x 1080 Scanning and Interface.*

SMPTE S17.392 (1995), *Proposed Standard for television, 1280 x 720 Scanning and Interface.*

ITU-R BT.601-4 (1994), *Encoding parameters of digital television for studios.*

A6.3. Compliance notation

As used in this document, *"shall"* or *"will"* denotes a mandatory provision of the standard. *"Should"* denotes a provision that is recommended but not mandatory. *"May"* denotes a feature whose presence does not preclude compliance, that may or may not be present at the option of the implementor.

HD MPEG VIDEO DECODER

APPENDIX B

APPLICATIONS BUS

Thomson Consumer Electronics, Inc.
Indianapolis, Indiana, USA

Revision No. 2.2

THESE DRAWINGS AND SPECIFICATIONS ARE THE PROPERTY OF THOMSON CONSUMER ELECTRONICS, INC. AND SHALL NOT BE REPRODUCED OR COPIED OR USED AS THE BASIS FOR THE MANUFACTURE OR SALE OF APPARATUS OR DEVICES WITHOUT WRITTEN PERMISSION.

TCE PROPRIETARY AND CONFIDENTIAL

B1. OVERVIEW

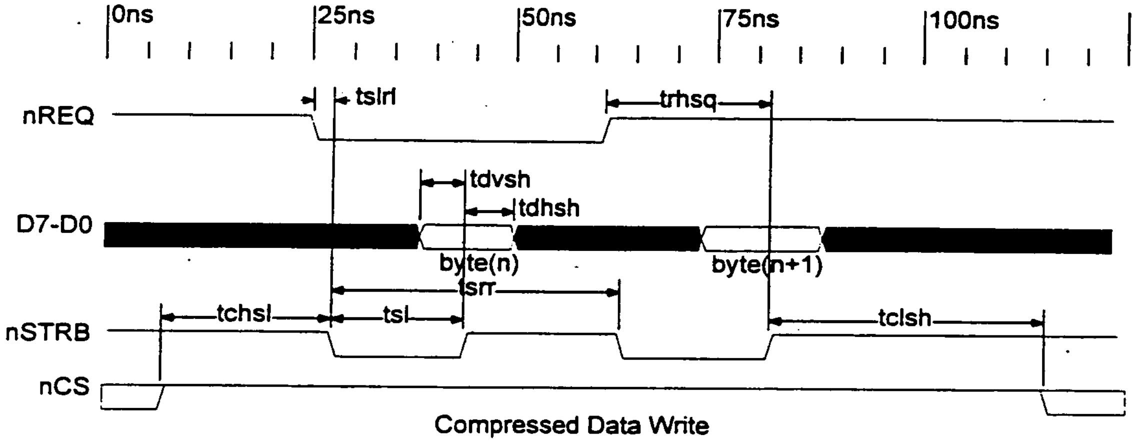
The Applications bus provides a path for compressed data transfer from the HOST to the compresed data interface as well as bi-directional HOST to register data path for control of the HD MPEG Video Deocder IC.

B2. APPLICATIONS BUS

The Applications Bus is an external manifestation of the internal "R-BUS". The conversion of the external Applications Bus to the internal "R-BUS" is accomplished through the internal "Microcomputer Interface".

B2.1. Compressed Data Transfer

The HD-MPEG IC must be capable of accepting compressed data at any rate up to 80 Mbits/sec (10 Mbytes/sec) averaged over any 188us, and longer, interval of time.



Compressed Data Write
Figure B1) Compressed data timing

G:\MPEG\HD-VIDEO\NEW_SPEC\DRAWINGS\COMPRES.TD

Row		Name	Formula	Min	Max	Margin	Comment
1	C	tslrl	[0,]	0		<2.46,>	nSTRB low from nREQ low
2	C	tdhsh	[5,]	5		<1.06,>	data hold from nSTRB high
3	C	tsrr	[35,]	35		<0.61,>	nSTRB repeat rate for nREQ low
4	C	tsl	[15,]	15		<1.12,>	nSTRB width
5	C	tdvsh	[5,]	5		<0.49,>	data valid from nSTRB high·
6	G	trhsq	[-5,40]	-5	40		nREQ high to nSTRB quit
7	C	tchsl	[20,]	20		<0.55,>	nCS high to nSBRB low
8	C	tclsh	[20,]	20		<13.53,>	nCS low from nCS high

Figure B2) Compressed data parameters

B2.2. Host to Applications Register Timing

B2.2.1. Host Write Cycle Timing

The following figures illustrate the compressed data transfer and host bus access timings.

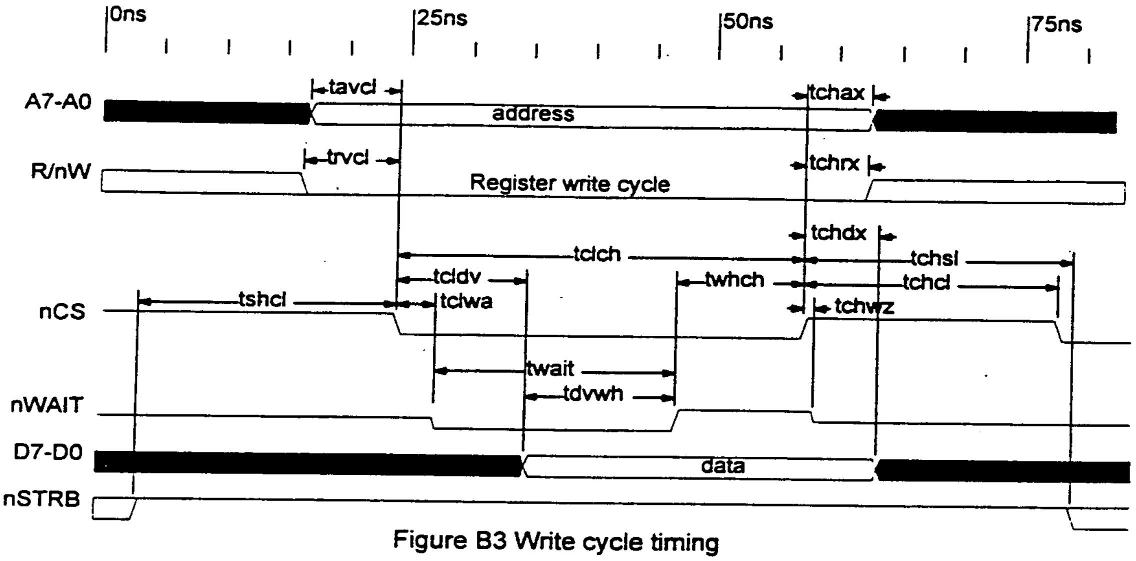


Figure B3 Write cycle timing

G:\MPEG\HD-VIDEO\NEW_SPEC\DRAWINGS\HOSTW.TD

Row		Name	Formula	Min	Max	Margin	Comment
1	C	trvcl	[5,]	5		<2.79,>	R/nW to nCW set-up time
2	C	tchax	[5,]	5		<0.31,>	address from nCS hold time
3	C	tchrx	[5,]	5		<0.03,>	R/nW from nCS hold time
4	C	twait	[,100]		100	<,80.39>	maximum wait time
5	C	tchcl	[20,]	20		<0.65,>	nCS high to nCS low
6	C	tavcl	[5,]	5		<2.26,>	address to nCS setup time
7	C	tchdx	[5,]	5		<0.97,>	data hold from nCS high
8	C	tchwz	[,5]		5	<,4.18>	nCS high to nWAIT off
9	C	tshcl	[20,]	20		<0.95,>	nSTRB high to nCS low
10	C	tchsl	[20,]	20		<1.88,>	nCS high to nSTRB low
11	C	twhch	[0,]	0		<10.47,>	nWAIT high to nCS high
12	C	tdvwh	[0,]	0		<12.27,>	data valid to nWAIT high
13	G	tclwa	[0,5]	0	5		nCS low to nWAIT active
14	G	tclch	[20,]	20			nCS low to nCS high
15	G	tcldv	[10,]	10			nCS low to data valid

Figure B4 Write cycle parameters

B2.2.2. Host Read Cycle Timing

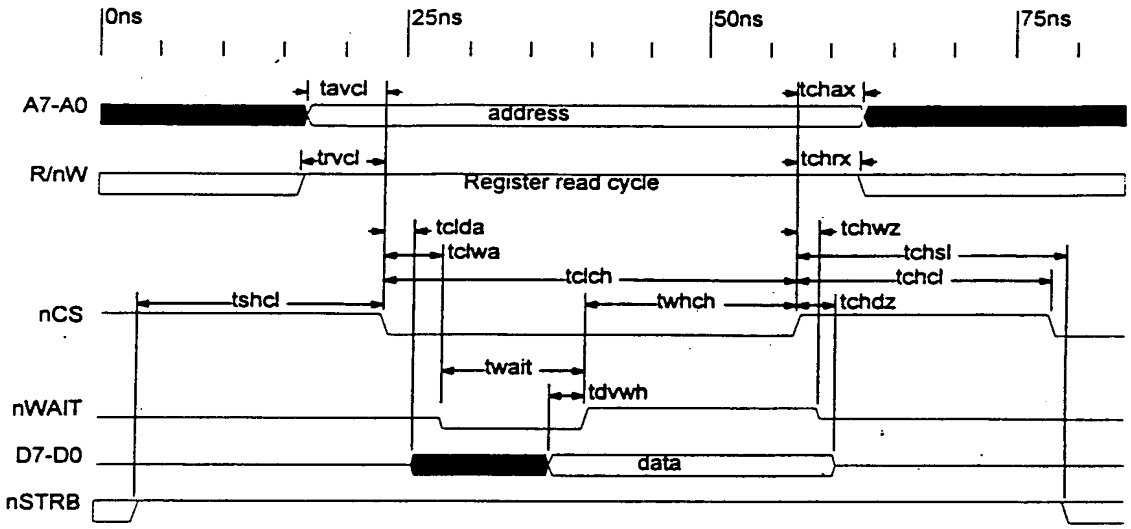


Figure B5) Read cycle timing

G:\MPEG\HD-VIDEO\NEW_SPEC\DRAWINGS\HOSTR.TD

Row		Name	Formula	Min	Max	Margin	Comment
1	C	trvcl	[5,]	5		<1.94,>	R/nW to nCW set-up time
2	C	tchax	[5,]	5		<0.31,>	address from nCS hold time
3	C	tchrx	[5,]	5		<0.03,>	R/nW from nCS hold time
4	C	twait	[0,100]	0	100	<11.64,88.36	maximum wait time
5	C	tchcl	[20,]	20		<0.65,>	nCS high to nCS low
6	C	tavcl	[5,]	5		<1.41,>	address to nCS setup time
7	C	tchwz	[0,5]	0	5	<1.76,3.24>	nCS high to nWAIT off
8	C	tshcl	[20,]	20		<0.10,>	nSTRB high to nCS low
9	C	tchsl	[20,]	20		<1.88,>	nCS high to nSTRB low
10	C	twhch	[0,]	0		<17.65,>	nWAIT high to nCS high
11	G	tclwa	[0,5]	0	5		nCS low to nWAIT active
12	G	tclch	[20,]	20			nCS low to nCS high
13	G	tchdz	[0,5]	0	5		nCS high to data off
14	C	tdvwh	[0,]	0		<2.92,>	data valid to nWAIT high
15	C	tclda	[0,]	0		<2.36,>	nCS low to data active

Figure B6) Read cycle parameters

HD MPEG VIDEO DECODER

APPENDIX C

DISPLAY INTERFACE

Thomson Consumer Electronics, Inc.
Indianapolis, Indiana, USA

Revision No. 2.2

THESE DRAWINGS AND SPECIFICATIONS ARE THE PROPERTY OF THOMSON CONSUMER ELECTRONICS, INC. AND SHALL NOT BE REPRODUCED OR COPIED OR USED AS THE BASIS FOR THE MANUFACTURE OR SALE OF APPARATUS OR DEVICES WITHOUT WRITTEN PERMISSION.

TCE PROPRIETARY AND CONFIDENTIAL

C1. OVERVIEW

The HD-MPEG IC will be used in television receiver applications and in set-top boxes. In a television receiver, the HD-MPEG IC will generate horizontal and vertical synchronizing signals which are used by the television deflection circuits. These synchronizing signals are generated by counting the appropriate line and frame periods using an externally supplied clock.

Although the HD-MPEG IC will receive and decode image sequences corresponding to many formats, a television receiver will likely use a common display format for all received formats. The display section in the HD-MPEG IC will contain horizontal and vertical resampling filters which are used to perform the format conversion.

The display section of the HD-MPEG IC must:

1 process 4:2:0 format images from the external memory; these images may be produced by the MPEG decode process or from the SD Pixel Port input. These images will be in 8x8 block format and must be converter to raster line images in the display section.
2 reformat the images under user control according to the procedures described in this document
3 output the reformatted images in 4:2:2 format in a rasterized manner as defined by the programmed raster generation parameters
4 overlay OSD into the output pixel stream (OSD generation described in another appendix)
5 provide an output pixel stream with luma pixel rates up to 81 MHz
6 generate horizontal and vertical drive signals for use by an external display device or accept external horizontal and vertical synchronization signals

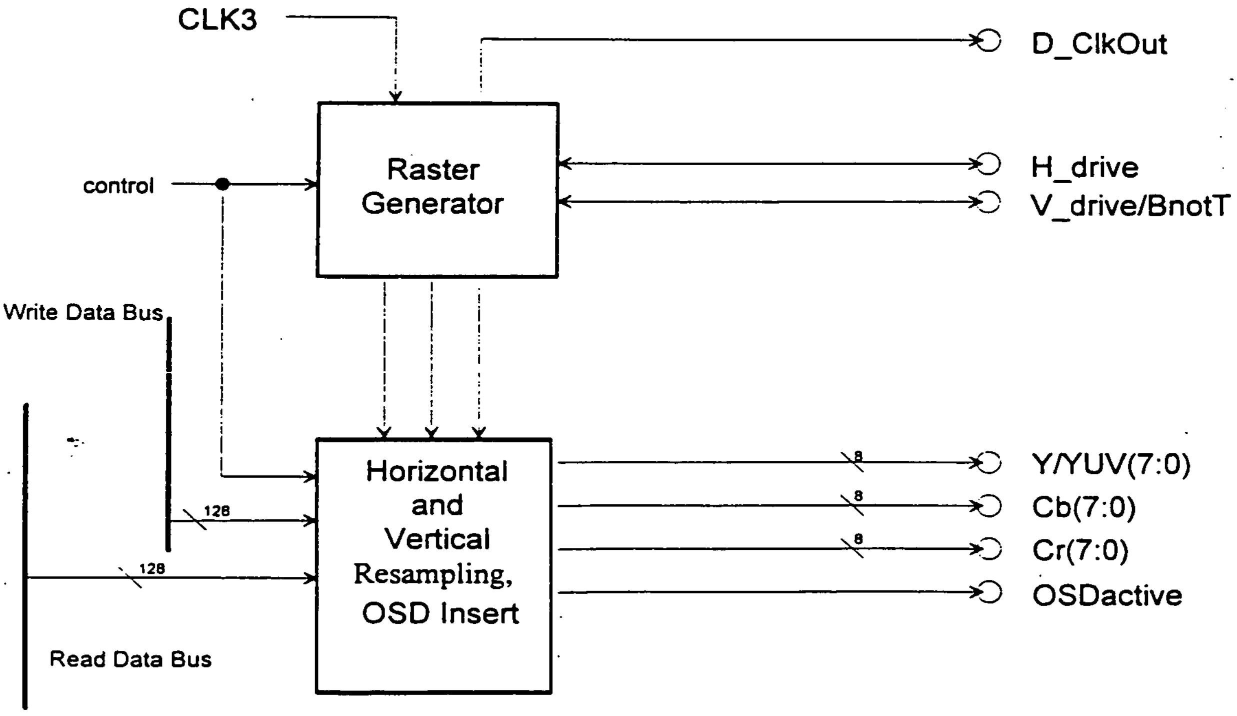
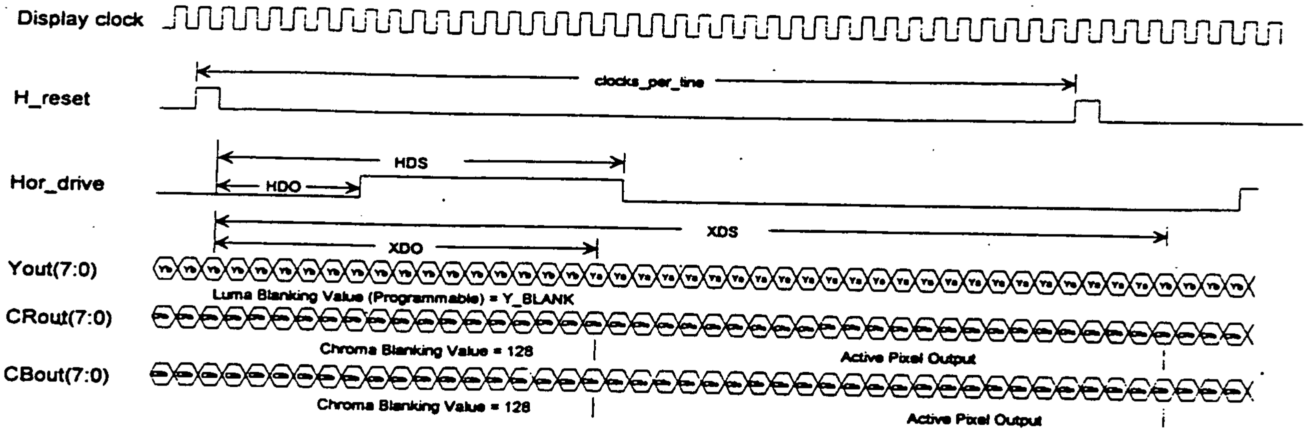


Figure C0 - HD-MPEG Display Interface

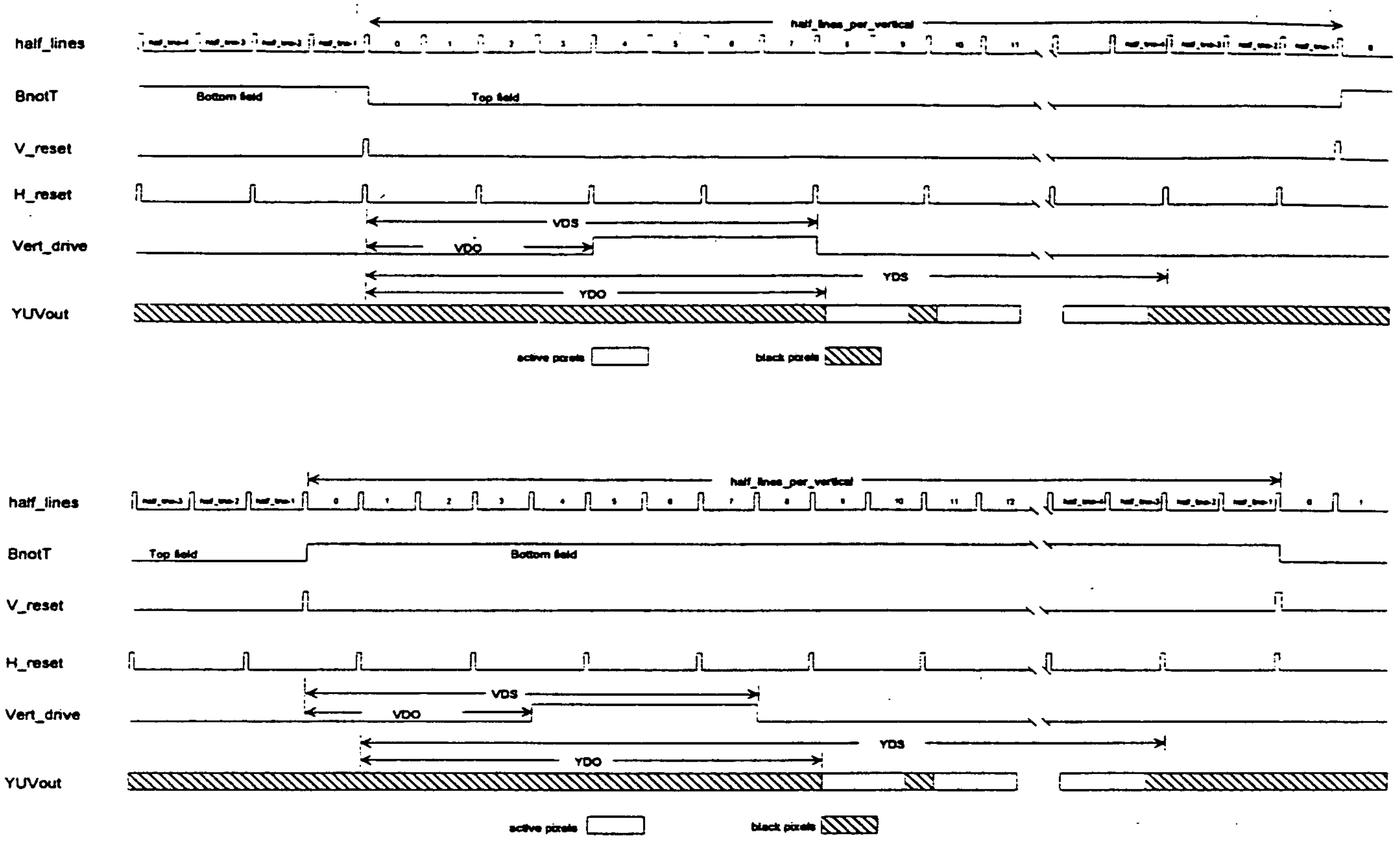
C2. RASTER GENERATOR

The raster generator counts display clock cycles and creates horizontal and vertical drive signals for use by an external display device as well as providing all the internal display related timing signals within the HD-MPEG decoder IC. The following figures illustrate the timing of the horizontal and vertical drive signals.



Horizontal Raster Timing

The vertical timing parameters must be double buffered and latched by frame sync (Bottom to Top field transition) as the half_lines_per_vertical count will be varied dynamically in some applications.



Vertical Raster Timing

C3. FORMAT CONVERSION

The HD-MPEG IC must be able to resample in both the horizontal and vertical directions; polyphase FIR filters are used to interpolate the desired pixels and lines. Required vertical processing also includes a line doubling operation. The following sections describe the required conversions.

C3.1. Display Decompression

Memory compression techniques are used to minimize the amount of external RAM required for MPEG processing (see appendix G for details). The compressed data, intended for display, is decompressed by two decompression blocks (one for luma, one for chroma) within the display spection (see Figure C1, or C4, or C6).

C3.2. Block to Line Conversion

The data format must be converted to raster scan lines for use by the rest of the display section. This is accomplished by writing an entire row of 8x8 blocks (luma) or 4x4 blocks (chroma) into the RAM and then reading out the data line by line. While the data is being read, new data is written into the location just read to minimize total memory required.

C3.3. Pan and Scan

In some cases the number of pixels needed to generate the display is fewer than the number stored in memory. An example of this is when a 16:9 aspect ratio image 720 pixels wide is to be displayed on a 4:3 display. In this case only 540 pixels of the original image are needed (although these 540 pixels may be sample rate converted to some greater number). 180 pixels from each line are discarded in this case. The discarded pixels may come from the right portion of the image, the left portion, or from a combination of the two.

A pan and scan vector is transmitted in the MPEG picture headers and indicates the more important part of the image.
When image cropping is required, this information can be used to choose which pixels/lines to discard.

The HD-MPEG IC must be able to start the display with a line and pixel other than the top left corner of the stored image. The offset must be programmable with the following resolution:

horizontal pan vector	1/16 pixel resolution
vertical pan vector	2 field line (4 frame line) resolution

The pan and scan vectors, as with all higher level MPEG information, must be accessible to the external micro controller.

C3.4. Horizontal Sample Rate Conversion

The horizontal sample rate converter must be able to support the following horizontal conversions:

Input Format	Output Format
352, 480, 544, 640, 720, 1280, 1920	1920
352, 480, 544, 640, 720, 960	960
352, 480, 544, 640, 720	720

The horizontal sample rate converter must support a maximum output pixel rate of 81 MHz.

maximum output pixel rate	81 MHz

The above table describes the luma formats. The corresponding conversions for chroma components must also be made including conversion from 4:2:0 to 4:2:2 format.

C3.5. Vertical Sample Rate Conversion

The vertical sample rate converter must be able to support the following vertical format conversions:

Input Format	Output Format
720 progressive	480 interlace; 480 progressive; 1080 interlace
1080 interlace	480 interlace; 480 progressive
240 CIF (240 lines at 30 Hz)	480 interlace; 480 progressive; 1080 interlace

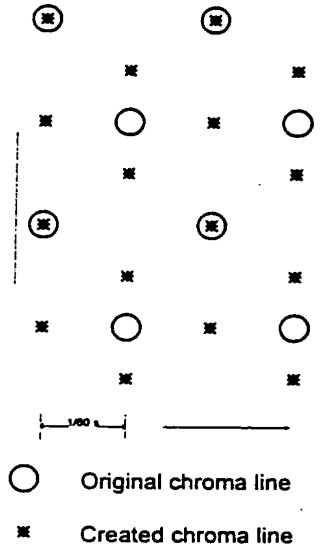
The luma vertical sample rate converter will be a 3 tap polyphase filter type, with bypass capability.
The above table describes the luma conversions. In all cases the input format has a 4:2:0 relationship between luma and chroma. In addition to the conversions listed above, 4:2:0 to 4:2:2 conversion must also be done for chroma. In the case of chroma, a 2 tap polyphase filter is used for the combined resampling and 4:2:0 to 4:2:2 conversion.

The following sections illustrate the various modes of operation in which the vertical re-sampler must operate.

C3.5.1. Conversion from 4:2:0 to 4:2:2 Format

In all cases some processing of chroma is required as video is stored in memory in 4:2:0 format and the display device expects 4:2:2 format data. Usually this chroma processing will be included with any other required vertical processing. The figure below illustrates the vertical/temporal relationship of input and output chroma lines when 4:2:0 to 4:2:2 only conversion is required (i.e. receive 480 interlace and display 480 interlace or receive 1080 interlace and display 1080 interlace).

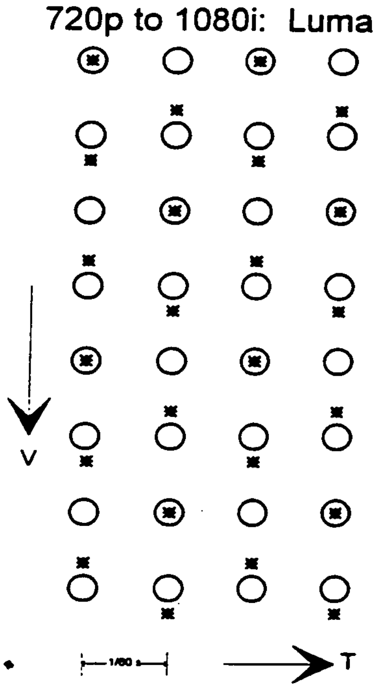
4:2:0 to 4:2:2 Field Based



The field based case is always used. In this case the even chroma lines (starting with 0) are used to generate the first or top field; the odd chroma lines are used to generate the second or bottom field.

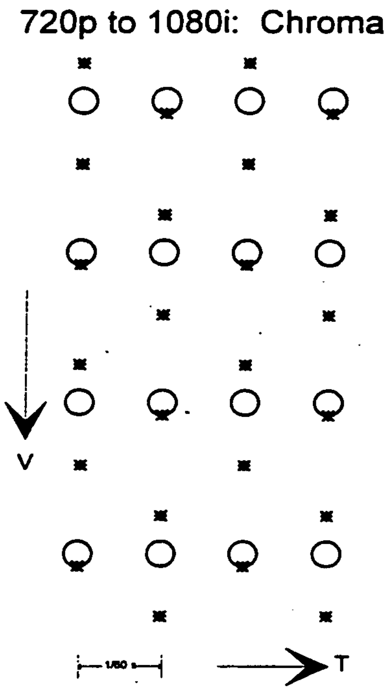
C3.5.2. Conversion from 720 Progressive to 1080 Interlace

The figure below illustrates the vertical/temporal relationship of input and output luma and chroma lines when the 720 progressive format is converted to 1080 interlace. The chroma case includes the 4:2:0 to 4:2:2 conversion.



○ source 720 progressive line

* target 1080 interlace line



○ Original 720p 4:2:0 chroma line

* Created 1080i 4:2:2 chroma line

Both the luma and chroma processing occurs only in the vertical direction. No temporal processing is used.

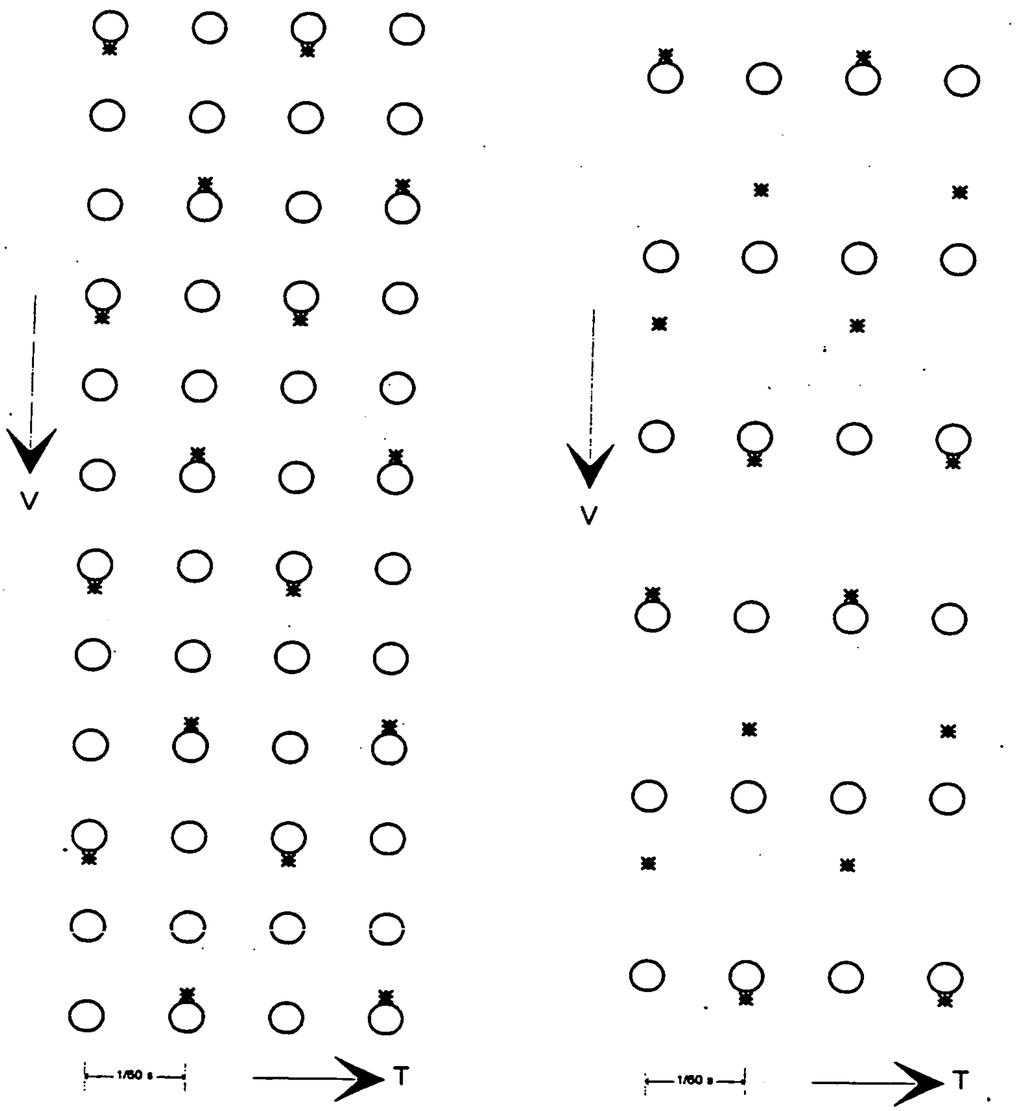
Note the input chroma is always frame based thus only frame based 4:2:0 to 4:2:2 conversion need be considered.

C3.5.3. Conversion from 720 Progressive to 480 Interlace

The figure below illustrates the vertical/temporal relationship of input and output luma and chroma lines when the 720 progressive format is converted to 480 interlace. The chroma case includes the 4:2:0 to 4:2:2 conversion.

720p to 480i: Luma

720p to 480i: Chroma



○ Original 720p luma line

✱ Created 480i luma line

○ Original 720p 4:2:0 chroma line

✱ Created 480i 4:2:2 chroma line

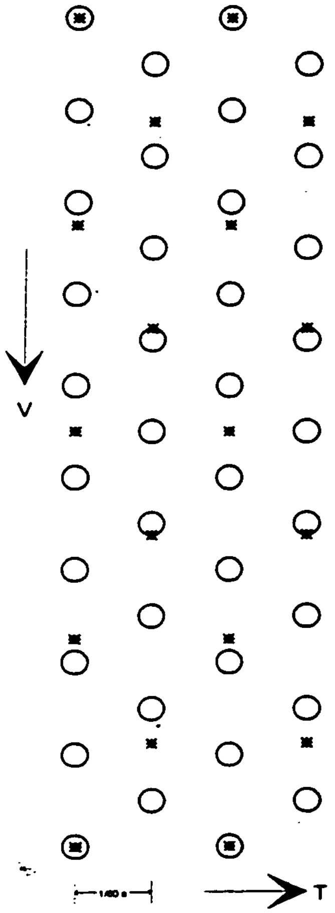
Both the luma and chroma processing occurs only in the vertical direction. No temporal processing is used.

Note the input chroma is always frame based thus only frame based 4:2:0 to 4:2:2 conversion need be considered.

C3.5.4. Conversion from 1080 Interlace to 480 Interlace

The figure below illustrates the vertical/temporal relationship of input and output luma and chroma lines when the 1080 interlace format is converted to 480 interlace.

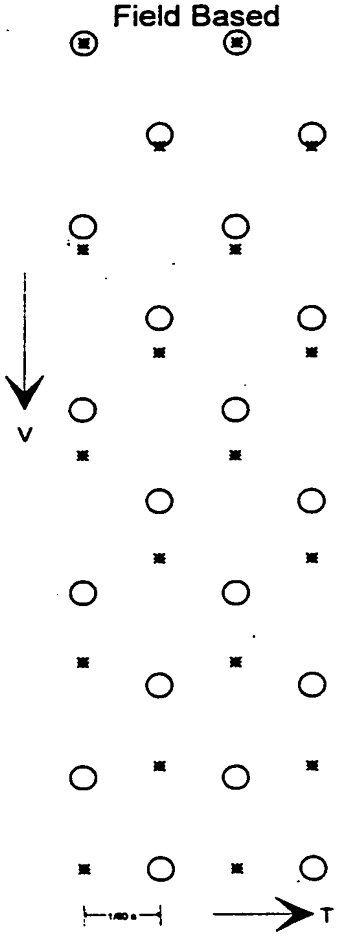
1080i to 480i: Luma



○ source 1080i luma line

✱ target 480i luma line

1080i to 480i: Chroma
Field Based



○ Original 1080i 4:2:0 chroma line

✱ Created 480i 4:2:2 chroma line

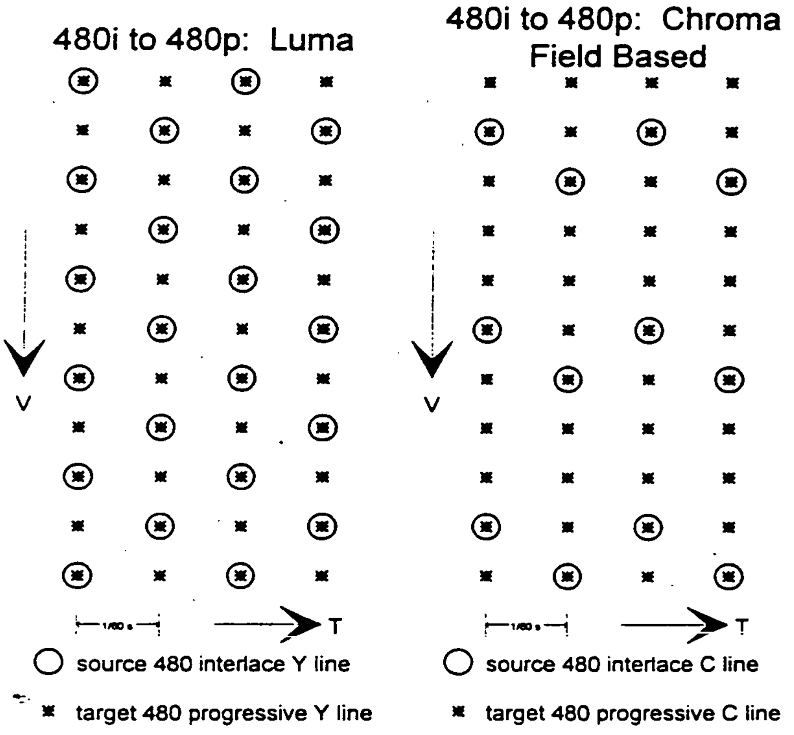
C3.6. Line Doubler

The HD-MPEG IC must have the capability of de-interlacing SD picture formats for the purpose of generating a 1125 total line (1080 active line) interlace output, or a 480 (active line) progressive output. The SD picture format has 480 active interlace lines. The output must have at least 480 active progressive lines (960 active interlace lines also acceptable?). The remaining active lines (1080 less 960) may be black.

Input Format	Output Format
480 interlace	480 progressive

The LMU (Linear Median Upconversion) de-interlacing algorithm as described in the HD-MPEG display section design specification is used.

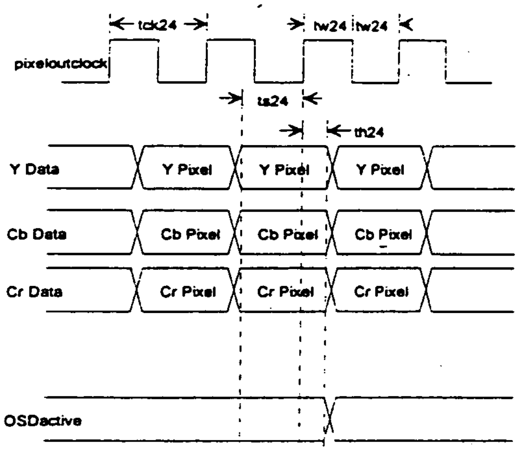
The following diagram illustrates the vertical/temporal relationship of lines before and after the deinterlacing process.

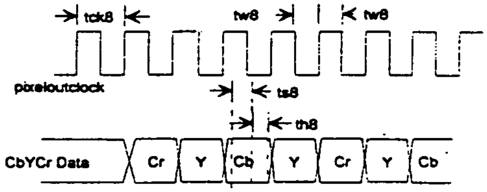

480i to 480p: Luma
480i to 480p: Chroma Field Based
V
1/60 s
T
○ source 480 interlace Y line
✱ target 480 progressive Y line
V
1/60 s
T
○ source 480 interlace C line
✱ target 480 progressive C line

Note in this case simple vertical direction only processing is not used; the required LMU algorithm uses information from neighboring fields to generate the new lines.

Note operation of the deinterlacing algorithm is only required for image sizes up to 720x480 interlaced (i.e. CCIR601 resolution). These images can originate from the MPEG decoding process or as input from the SD pixel port. **The HD-MPEG IC is not required to deinterlace high definition display formats.**

C4. INTERFACE TIMING





parameter	min	units
tck24	12	ns
tw24	4	ns
ts24	3	ns
th24	2	ns

parameter	min	units
tck8	35	ns
tw8	14	ns
ts8	6	ns
th8	3	ns

C5. DESIGN

This design specification describes the design which will implement the functions of the display section described in the HD-MPEG Specification and Appendices C and D for the display and OSD sections.

The top level diagram of the display section is shown in Figure C 1. In this diagram, video and motion data is read from the 128-bit internal bus, is decompressed (when necessary), convertered from 8 by 8 (or 4 by 4 for chroma) block format to raster line format, processed by horizontal and vertical sample rate converters (and in some modes the LMU line doubler). The final calculated output video is mixed with the bit-map data in the OSD section and then becomes the data output of the IC. The display section contains the Raster Generator section which generates the vertical and horizontal drive signals available to control the display as well as pixel, line, half-line counts, and horizontal reset and vertical reset counts available internally in the IC. The display section also contains three, RAM blocks which are used for block-to-line conversion and line delays for the vertical sample rate converter (V - SRC) and the LMU line doubler blocks.

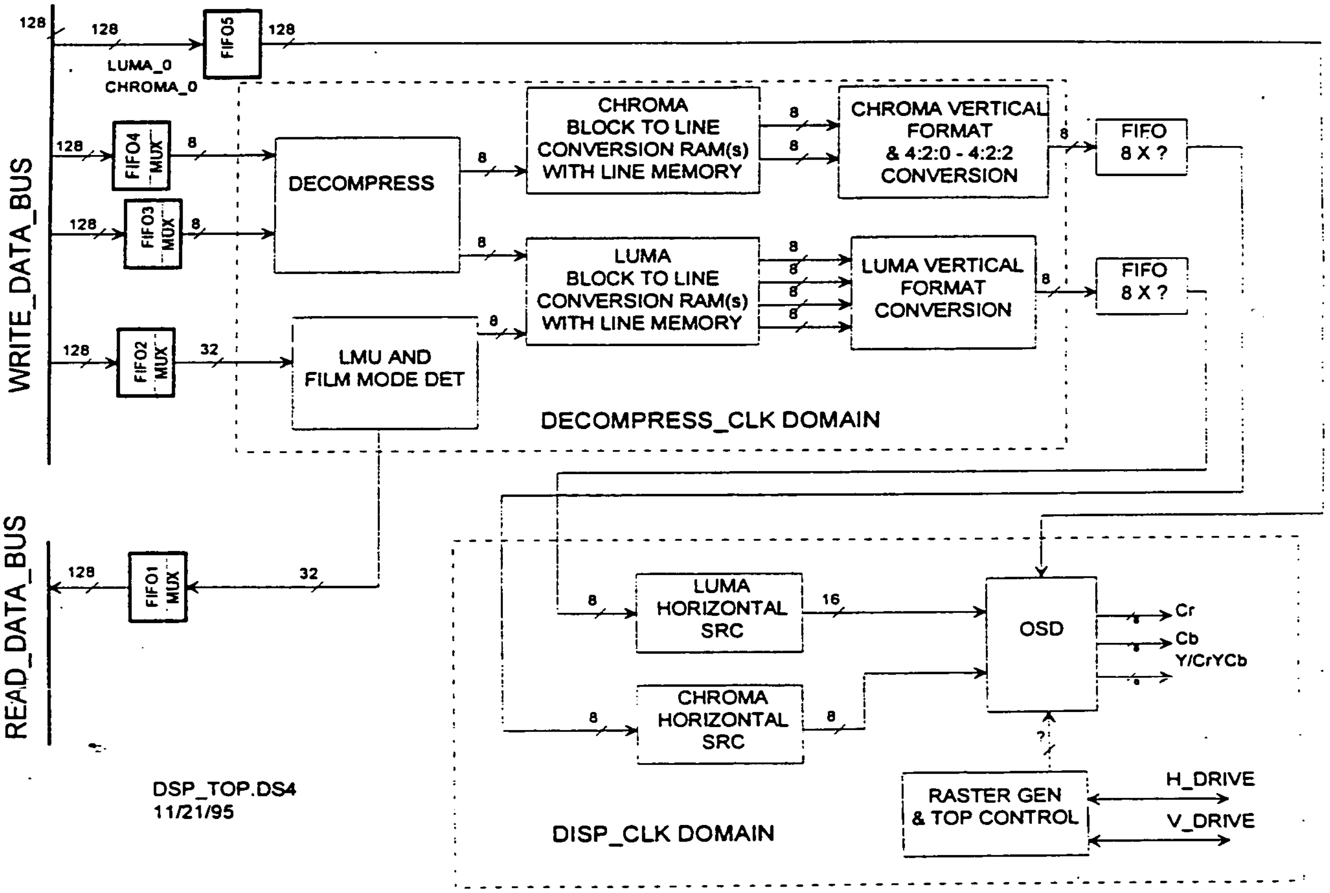


Figure C 1. Display Top Level

As can be seen in Figure C 1 (above) the display section is made up of two different clock domains, the DECOMPRESS_CLK domain, and the DISP_CLK domain. The DECOMPRESS_CLK domain contains all the functions which must interface synchronously with the block-to-line conversion RAM and must run at 40 -

81Hz clock rates to achieve the desired bandwidth. The DISP_CLK (display clock) domain contains the functions which must run synchronously with the final output, and may run at clock rates from 27 - 81 MHz. Note that in some applications the two clocks may be the same, in other applications they will be different. Video data passing between the two clock domains goes through FIFO's (one each fo luma and chroma) with the read request for the FIFO coming from the H-SRC controller.

C5.1. FIFOs & LMC

There are 5 FIFOs which interface with the LMC in the HD-MPEG display section (as shown in Figure C 1), 4 are on the INPUT_DATA_BUS FIFOs and 1 is on the OUTPUT_DATA_BUS FIFO. There will be two main components to each FIFO:

(1) The actual FIFO (to be delivered by ST) is 128 bits wide by **about?** 16 or 32 words deep (the exact depth to be determined later). The FIFO data interface to the display section is 8 bits wide and must run at up to 81 MHz. The FIFO data interface to the data bus is 128 bits wide, and must run at **approximately** 50 MHz.

(2) The FIFO control logic (designed by TCE at ST): This interfaces with read and write acknowledge and request signals from the display and LMC blocks, keeps track of the amount of data in the FIFO and carefully handles the asynchronous interface between the "bus" end of the FIFO which uses the same clock as the data bus and the "display" end of the FIFO which uses the display clock. Since the control logic is in the display section, the circuitry actually running off of the "bus" clock will be minimized. The FIFO control to display interface (for the input FIFO's) consists of the following:
RST to the FIFO controller - signalling FIFO to clear and re-fill.
DATA_REQ to the FIFO controller - signalling FIFO to pump out data one clock later.
DATA from the FIFO controller - one clock after DATA_REQ goes high.

C5.1.1. Local Memory Interface Requirements

The vertical format conversion process places several new requirements on the local memory controller (LMC), since display data is not always required in a continuous, uniform manner. Lacking a detailed definition of the LMC operation, the following model for the interface has been assumed.

For each of the video data paths (FIFOs 3,4,5,6,7), there exists a memory pointer, an active line register, an upper field start register, and a lower field start register. The upper and lower field start registers will be set via the host bus to the number of the first frame line to be read, where 0 corresponds to the first active line of the frame. At the beginning of each upper or lower field, data from the corresponding field start register is transferred to the active line register.

At the start of each display line, the LMC uses the contents of the active line register to calculate a physical memory address (using the lsb to select the appropriate field, if fields are stored separately). The result is loaded into the memory pointer. The FIFOs are then reset and filled with data corresponding to the specified display scan line. During the line interval additional data is fetched as required by the FIFOs, modifying the memory pointer in the process. Sometime during the same line interval the active line register is incremented by n, where n = 0,...,7. The vertical format converter provides a new value of n for each line for each FIFO.

When performing format conversion, FIFO2 is used to access control words for the vertical format converter. Each control word is contained within a 128 bit word of the local memory. Separate upper and lower field start registers are required for this function as well, but only one 128 bit word is used for each display line. A signal from the vertical format converter will signal the LMC to fetch the next line's control word from either (1) the next successive memory location or (2) from the location specified by the field start register.

C5.1.2. FIFO and LMC Interface

The attached diagrams illustrate a proposal for the interface for the FIFO's in the Display section (not including the OSD FIFO which is really a special case). The diagrams also show signals going ito and out of a clock domain transfer block.

Figure C 2 shows a Memory Read FIFO block and the connections with the block entitled CLOCK DOMAIN TRANSFER AND BUS DECODER. The left side of the blocks represents the LMC clock domain and contains signals coming from or going to the LMC. These signals are synchronous with LMC_CLK (Local Memory Controller clock). The right side of the blocks represents the Display domain and contains signals coming from or going to the Display section. These signals are synchronous with DISP_CLK (Display clock).

C5.1.3. Memory Read FIFO

The Memory Read FIFO and associated controller is shown enclosed by the dashed lines in Figure C 2. This FIFO occurs 6 times in the display section (as FIFOs number 2,3,4,5,6,7 in the HD-MPEG DISPLAY SECTION DESIGN SPECIFICATION).

The LMC clock domain signals for this block are:
DATA_BUS(127:0) - the internal data bus, controlled by the LMC,
LTE_HALF_FULL_LMC - the less than or equal to half-full flag for the FIFO (to the LMC),
LMC_CLK - the LMC clock.

The display section clock domain signals are:
VIDEO_TO_DISPLAY(7:0) - the 8-bit, de-multiplexed data bus,
REQ - the data request signal to the FIFO controller,
H_RST - the horizontal reset signal (from the Raster Gen block in the display section),
DISP_CLK - the display clock,
PSV.H_DISP(8:0) - the horizontal pan vector (integer component).

Note that we propose the Display section to FIFO interface be synchronous with DISP_CLK , and the clock is not gated. Any gating of DISP_CLK we propose be done within FIFO_CNTL (FIFO controller block).

Figure C 2 also shows a timing diagram of the signals in the display clock domain which interface with the FIFO and FIFO controller. After H_RST = 1, there is some minimum time (not yet determined) before data can be requested (REQ = 1). If REQ = 1 for a period of time, for example 3 clock periods as shown in Fig. 2, then 3 sequential values of VIDEO_TO_DISPLAY are valid with a delay of about 1 to 3 clocks (again, not yet determined). The REQ signal will be going high and low with some pattern, and the valid data on the VIDEO_TO_DISPLAY bus will follow that same pattern with the as yet undefined 1 to 3 clock delay.

C5.1.4. Clock Domain Transfer and Bus Decoder

This block is also shown in Figure C 2. It transfers signals from the Display clock domain to the LMC domain and also decodes bus registers needed by both the LMC and Display sections. The Display clock domain signals transferred to the LMC clock domain are:
H_RST_LMC - LMC domain version of H_RST (see above);

V_RST_LMC - LMC domain version of V_RST, the vertical reset from the Raster Generator block of the display section;
RPT_MODE(2:0)_LMC - notifies the LMC which line to fetch from memory on H_RST: RPT_MODE=0 => repeat the line, RPT_MODE=1 => fetch the next line, RPT_MODE=2 => skip a line (if currently completing line N, then fetch line N+2),
RPT_MODE=K => (if currently completing line N, then fetch line N+K)(K= 0,...,7).

Since many of the modes of the display section require that the LMC either repeat or skip lines during reading, we propose that the all of the FIFO_CNTL blocks be identical (for the basic memory read FIFO) and all FIFO_CNTL blocks include the repeat mode function (RPT_MODE).

Host bus registers decoded and transferred to the LMC clock domain are:
PSV.H_LMC(8:0) - the integer component of the pan/scan vector.

Host bus registers decoded and transferred to the Display clock domain are:
PSV.H_DISP(8:0) - the integer component of the pan/scan vector,
LSO_DISP - the luma sub-pixel component of the pan/scan vector,
CSO_DISP - the chroma sub-pixel component of the pan/scan vector.

We propose that this block be designed by TCE and included in the display section.

C5.1.5. Memory Write FIFO

The memory write FIFO is shown in Figure C 3. This is FIFO1 as shown in the HD-MPEG DISPLAY SECTION DESIGN SPECIFICATION.

The LMC clock domain signals for this block are:
DATA_BUS(127:0),
GTE_HALF_FULL_LMC - the greater than or equal to half-full flag for the FIFO (to the LMC),
LMC_CLK - the LMC clock,

The display section clock domain signals are:
VIDEO_FROM_DISPLAY(7:0) - the 8-bit, motion video signal,
WR_EN - the write enable signal for the FIFO controller,
H_RST, DISP_CLK.

Note that we propose that the interface between the Display section and the Memory Write FIFO also be fully synchronous with DISP_CLK.

Figure C 3 also shows a timing diagram of the signals in the display clock domain which interface with the FIFO and FIFO controller. After H_RST = 1, there is some minimum time (not yet determined, but should be very short - hopefully only a few clocks) before data can be written (WR_EN = 1). If WR_EN = 1, valid data are ready at the DEMUX input of the FIFO.

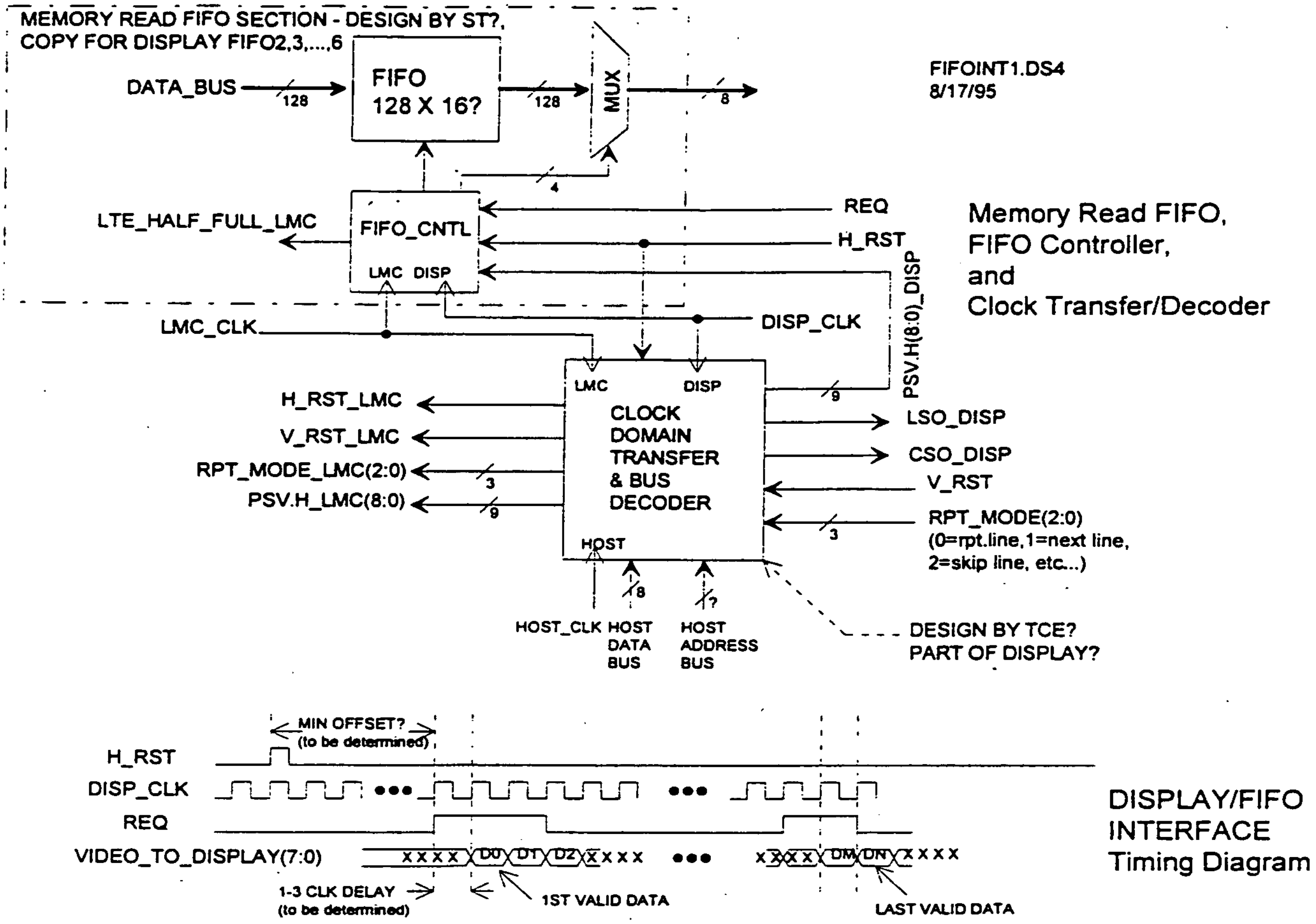


Figure C 2. Memory Read FIFO Interface

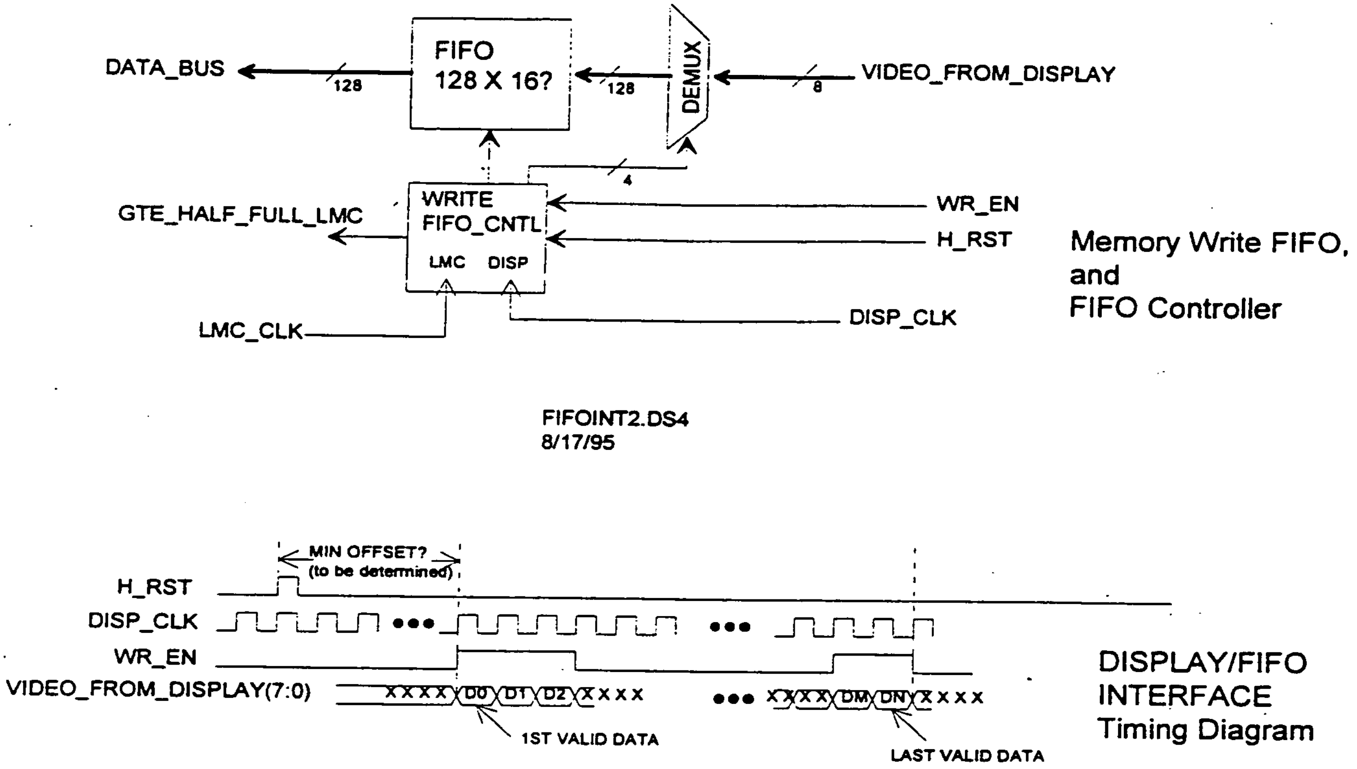


Figure C 3. Memory Write FIFO Interface

C5.2. Display Decompress

Memory compression techniques are used to minimize the amount of external RAM required for MPEG processing (see appendix G for details). The compressed data, intended for display, is decompressed by two decompression blocks (one for luma, one for chroma) within the display spection (see Figure C1, or C4, or C6). To handle the bandwidth requirements, the luma decompression block actually decompresses two 8x8 blocks simultaneously and delivers the decompressed blocks with the pixels interleaved together. The decompress block function may have overhead associated with the decompression function which could be 2 overhead clocks cycles for 64 pixels processed (e.g. takes 66 clocks to handle 64 pixels). The chroma decompression decompresses 4 by 4 blocks each of Cr and Cb.

C5.3. Block to Line Conversion

The memory compression technique encodes the luma in blocks of 8 by 8 pixels, and the chroma in blocks of 4 by 4 pixels per component. The decompress blocks interleave two sequential blocks (e.g. two 8x8 luma blocks), and the data passed out of the display decompress blocks is a sequence of pixels from two interleaved blocks. After the serial-to-parallel converter (S-P), implemented using a FIFO/demux, the data is de-interleaved, and is ready for a write to the RAM.

The data format must be converted to raster scan lines for use by the rest of the display section. This is accomplished by writing an entire row of 8x8 blocks into the RAM and then reading out the data line by line. While the data is being read, new data is written into the location just read to minimize total memory required. This results in the data being "scrambled" in the memory, but a straightforward address update algorithm keeps track of the address pointer increment.

For both luma and chroma, the entire block-to-line conversion RAM and control system runs at 27 MHz (1/3 of the decompress clock rate).

C5.3.1. Luma Decompress and Block to Line Conversion

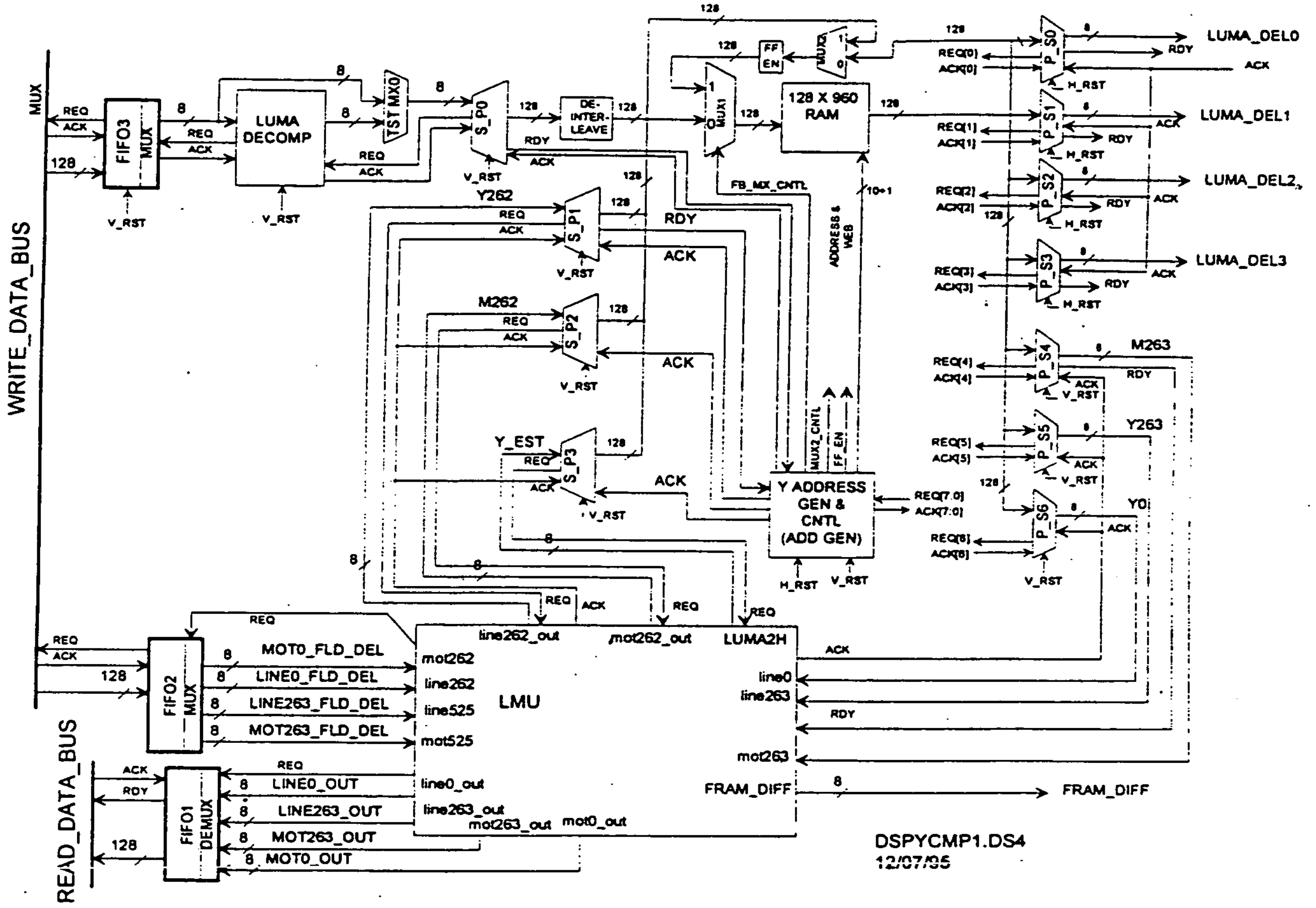


FIGURE C 4. DISLAY LUMA DECOMPRESS AND BLOCK CONVERSION RAM

As shown in Figure C4, the luma channel has a RAM used for block-to-line conversion and line memory. The memory compression technique encodes the luma in blocks of 8 by 8 pixels. The decompress block.interleaves the two 8x8 blocks, and the data passed out of the luma display decompress block is a sequence of pixels from two interleaved 8x8 blocks. After the serial-to-parallel converter (S-P), the data is de-interleaved, and one 128 bit-wide (16 pixels, 8 bits/pixel) word is ready for a write to the RAM.

In some modes of operation, the data passes through one serial-to-parallel converter (S_P0), the block-to-line RAM, and one parallel-to-serial converter (P_S0). The entire RAM may be used for block-to-line conversion. In this mode, the address generator and control block (Y ADD GEN) is really only concerned with the block-to-line conversion process. In other modes, up to four parallel-to-serial converters (P_S0 ... P_S3) are used to send data to the vertical filter. In these modes, the entire RAM is not used for block-to-line conversion; enough memory is available to store up to four lines of luma video. The ADD GEN must perform writes to the line memory space in the RAM, reads from the line memory space, and control the four parallel-to-serial converters. When the LMU block is used, additionally two more lines of data are stored in the RAM, and a total of 3 I/O streams are added going between the RAM and the LMU block.

The two MUX's in the a feedback loop around the RAM allow a RAM read to be recirculated for a RAM write (used to allow writing of block-to-line conversion output in to the line memory space) and also allow data just written into the memory to be written a second time into a different location (for LMU mode).

The entire data flow through the circuitry is based on either a data request (REQ) and acknowledge (ACK); or data is ready (RDY) and acknowledge (ACK). As can be seen in Figure C4, different FIFO's in different applications make different uses of these structures.

> The input of all FIFO's internal to the display are controlled as follows:
> When the FIFO is less than half full, the REQ goes high. When the FIFO receives an ACK from the source, it latches in the data.
>
> The output of all S_P and P_S FIFO's are controlled as follows:
> When data is available to be read from the FIFO, the RDY signal goes high. When the FIFO receives an ACK, data is clocked out of the FIFO.
>
> Note that all of the S_P and P_S FIFO's are fully synchronous in that the input and output clocks are the same.

C5.3.2. Luma Ram Address Generator (Y_ADD_GEN)

The luma address generator, shown in Figure C 5, consists of five concurrent processes and an arbitrator. The first four processes (Write block to line memory, Read block to line memory and LMU memory, Write output and LMU memory, Read output memory) monitor and control the status of the various sections of the RAM, as well as the S_P and P_S FIFO's, the feedback muxes and flip-flop. Based on the need to read from or write to the appropriate sections of RAM, each process requests RAM access (REQ/RDY), and is granted access (ACK) by an arbitrator. The fifth process is a refresh process which has programmable refresh rates and refresh address increments. The refresh process must have priority over the other processes; otherwise the other four process may have equal priority.

The job of the arbitrator is to accept requests for read/write to the from the five processes and to acknowledge (allow a write/read to occur) them on a priority basis. The refresh gets highest priority; all other processes can probably have randomly assigned priorities. Concurrent with the acknowledgment, controls must be sent to the MUX shown in Figure C5 to allow the MEMORY_ADDRESS and WEB control of the bus going to the RAM.

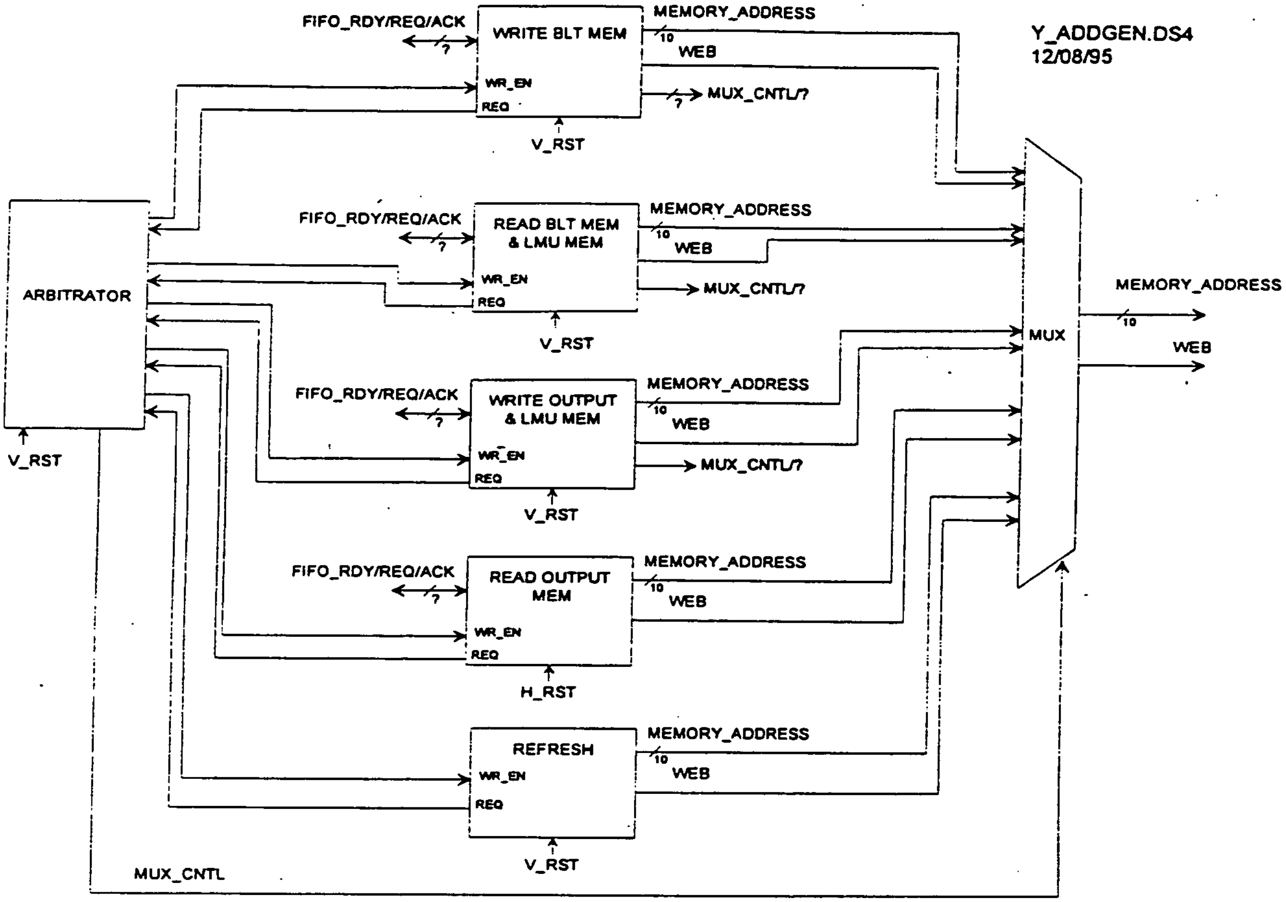


FIGURE C 5. Luma Ram Address Generator

C5.3.2.1. Write to Block-to-Line Conversion Memory

This process accepts a 128 bit word from the FIFO connected to the output of the memory decompress block. The word corresponds to 16 horizontally adjacent pixels from a pair of 8 x 8 blocks. The word is written to the block-to-line conversion portion (bottom) of memory, which stores one complete row of 16 x 8 block pairs. Address calculations make use of a parameter called BTL_Modulus, which has the value (8 x BlockPairs_per_Line - 1); this is numerically equal to the value of the highest address used.

This process is reset by V_RST, which also resets a BTL_Data_Valid flag. At the beginning of each row of blocks, the write address is set to zero; after each word is written, an increment value (given by BTL_delta) is added; the result is then reduced modulo BTL_Modulus. This algorithm would result in a value of zero for the last word of the block row; when this is detected a BTL_Write_Last flag is set, and word is stored at an address equal to BTL_Modulus instead. The memory is completely filled once at the start of each field, at which point the BTL_Data_Valid flag is set; after that, a word is not written until the data previously stored at that location has been read by the block to line read process. This is accomplished by inhibiting this write process when the write address matches the address being used by the read process.

The value of BTL_delta changes for each row of blocks processed. The first row of blocks in a field is written using a value of one. For each subsequent row, a new value of BTL_delta is calculated by multiplying the previous value by 8 and reducing the result modulo BTL_Modulus; this same value is used when calculating addresses for both reading the previously stored data and writing the new data.

The following resources are used by this process:

Context Registers: none

External Interfaces: BlockPairs_per_Line(6:0) // equal to picture width in macroblocks

BTL_Modulus(9:0) // equal to 8 * BlockPairs_per_line - 1

BTL_Read_Address(9:0) // present block-to-line read address

BTL_Write_Enable // grants access to memory (from arbitrator)

DCD_FIFO_RDY // ready line from the DeCompress Data FIFO

DCD_FIFO_ACK // acknowledge to the DeCompress Data FIFO

Mux1_Control // selects the FB Latch out as memory input source

Memory_Address(9:0) // Address bus to display RAM

Working Registers: BTL_Write_Address(9:0) // present block-to-line write address

BTL_delta(9:0) // conversion delta value for this row of blocks

BTL_delta_next(12:0) // delta value for the next row of blocks

BTL_Write_Last // ready to write last word of a block row

BTL_Data_Valid // flag to indicate that the BTL memory is full

BTL_Write_Request // request to arbitrator for access to memory

Temporary Variables: Temp1(10:0), Temp2(10:0), Temp3(12:0) // used in address calculation

Process Description (executes every potential memory cycle)

```
Mux1_Control = DCD_FIFO_ACK = 0;                                      // default conditions
if ( V_RST == 1 ) {                                                   // Initialize on vertical reset
        BTL_Write_Address = BTL_Data_Valid = BTL_Write_Last = BTL_Write_Request = 0;
        BTL_delta = 1;
        BTL_delta_next = 8;
}
else if ( BTL_Write_Enable == 1 ) {                                   // access to write a word has been granted
        BTL_Write_Request = 0;                                        // disable request
        Memory_WEB = 0;                                               // enable memory for writing
        Memory_Address = BTL_Write_Address;                           // block-to-line write
        DCD_FIFO_ACK = 1;                                             // enable DCD FIFO to output data
        Temp1 = BTL_Write_Address + BTL_delta;                        // calculate next BTL write address
        Temp2 = Temp1 - BTL_Modulus;                                  // subtract modulus
        if ( BTL_Write_Last == 1 ) {                                  // writing last word of row
                BTL_Write_Last= 0;
                BTL_Write_Address = 0;                                // reset pointer to start
                BTL_Data_Valid = 1;                                   // indicate that the memory has been filled
                BTL_delta = BTL_delta_next;                           // update delta value
                BTL_delta_next = 8 * BTL_delta_next;                  // calculate unreduced next delta value
        }
        else if ( Temp2 == 0 ) {                                      // address calculation result is zero
```

```
                BTL_Write_Last = 1;                              // this will be the last write for the block row
                BTL_Write_Address = Temp1;                                          // set to BTL_Modulus
        }
        else if ( Temp2 < 0 ) BTL_Write_Address = Temp1;         // subtraction result negative, don't use
        else BTL_Write_Address = Temp2;                                         // use subtraction result
}
                                                  // Otherwise, see if write cycle needs to be requested
else if ( ( BTL_Data_Valid == 0 || BTL_Write_Address != BTL_Read_Address ) && DCD_FIFO_RDY == 1 )
        BTL_Write_Request = 1;
                                                   // Otherwise, reduce next delta value if necessary
else {
        temp3 = BTL_delta_next - BTL_Modulus;
        if ( temp3 >= 0 ) BTL_delta_next = temp3;
}
```

`5.3.2.2. Read from Block-to-Line Conversion and LMU Memories

When no output line memories are in use (e.g. 1920 x 1080 mode), this process reads a 128 bit word from the block-to-line conversion portion (bottom) of memory and stores it in output FIFO number 0 whenever new data is requested and available. When output line memories are being used and LMU mode is not active, the word is written to the feedback latch instead. The new read address is then calculated using a BTL_delta value obtained from the block-to-line write process.

When in LMU mode, the process reads the same block-to-line word but stores it in the Y0 FIFO; the address calculation is the same as in non-LMU operation. Additionally, the two LMU line memories are read, and their data words are transferred to the Y525 and M525 FIFOs. The LMU function requires Y0, Y525, and M525 pixel data simultaneously, so every time Y0 data is read, an access request is generated to read Y525 data, which then requests an access to read M525 data. The two LMU line memories are written alternately and read alternately, so to simplify addressing their locations in memory are interleaved; LMU line memory 0 occupies even memory address locations and line memory 1 occupies odd addresses.

This process is reset by V_RST. It is synchronized with the write of the block-to-line conversion by preventing the write process from proceeding whenever the write address is equal to the read address generated by this process. A BTL_Data_Valid flag indicates that the block-to-line memory has been fully written at least once; at the start of each field, reading is inhibited until this flag is high.

The following resources are used by this process:

Context Registers: None

External Interfaces: BlockPairs_per_Line(6:0) // equal to picture width in macroblocks

LMU_Line0_Start(9:2) // start address of LMU line memories (from LMU write)

BTL_Write_Address(9:0) // present block-to-line write address (from BTL write)

Number_Output_Line_Mem(2:0) // number of output memories used (0 to 4)

BTL_Read_Enable // grants access to memory (from arbitrator)

Y0_FIFO_REQ // request line from the Y0 FIFO

Y0_FIFO_ACK // acknowledge to the Y0 FIFO

Y263_FIFO_ACK // acknowledge to the Y263 FIFO

M263_FIFO_ACK // acknowledge to the M263FIFO

Mux1_Control // selects the FB Latch out as memory input source

Mux2_Control // selects the memory input as FB Latch input

FB_Latch_Enable // enable the feedback latch to latch data

BTL_delta(9:0) // conversion delta value for this row of blocks (from BTL write)

BTL_Data_Valid // flag to indicate that the BTL memory is full (from BTL write)

BTL_Modulus(9:0) // equal to 8 * BlockPairs_per_line - 1 (from BTL write)

LMU_Mode // indicates that LMU mode is in effect

Memory_Address(9:0) // Address bus to display RAM

Working Registers: BTL_Read_Address(9:0) // present block-to-line read address

LMU_Read_Address(9:0) // present LMU line memory read address

LMU_Read_Word_Count(6:0) // words written in present line

BTL_Read_Last // ready to read the last word of a block row

FB_Latch_Valid // indicates that the latch contents are valid

BTL_Read_Request // request to arbitrator for access to memory

LMU_Read_Type(1:0) // specifies type of data transfer when in LMU mode

Temporary Variables: Temp1(10:0), Temp2(10:0) // used in address calculation

Process Description (executes every potential memory cycle)

```
Mux1_Control = Mux2_Control = Output_Read_FIFO_ACK(0) = 0;                // default conditions
Y263_FIFO_ACK = M263_FIFO_ACK = M263_FIFO_ACK = FB_Latch_Enable = 0;
if ( V_RST == 1 ) {                                                   // Initialize on vertical reset
        BTL_Read_Address = 0;
        LMU_Read_Address = LMU_Line0_Start * 4;
        LMU_Read_Word_Count = 0
        LMU_Read_Type = 0;
        FB_Latch_Valid = 0;
        BTL_Read_Last = 0;
        BTL_Read_Request  = 0;
}
else if ( BTL_Read_Enable == 1 ) {                        // access to read a word has been granted
```

```
        Memory_WEB = 1;                                                  // enable memory for reading
        if ( LMU_Mode == 0 || LMU_Read_Type == 0 ) {
                Memory_Address = BTL_Read_Address;                       // block-to-line read
                Temp1 = BTL_Read_Address + BTL_delta;                    // calculate next BTL read address
                Temp2 = Temp1 - BTL_Modulus;
                if ( BTL_Read_Last == 1 ) {                              // start new block row
                        BTL_Read_Last = 0;                               // reset pointer to start
                        BTL_Read_Address = 0;
                }
                else if ( Temp2 == 0 ) {
                        BTL_Read_Last = 1;                               // this will be the last read for the block row
                        BTL_Read_Address = Temp1;
                }
                else if ( Temp2 < 0 ) BTL_Read_Address = Temp1;
                else BTL_Read_Address = Temp2;
                if ( Number_Output_Line_Mem == 0 ) {                     // no output line memories used
                        Output_Read_FIFO_ACK(0) = 1;                     // Enable FIFO
                        BTL_Read_Request = 0;                            // no more cycles for now
                }
                else if ( LMU_Mode == 0 ) {                              // not LMU mode
                        Mux2_Control = 0;                                // select RAM output as latch source
                        FB_Latch_Enable = 1;                             // enable latch to capture data
                        FB_Latch_Valid = 1;                              // latch data will be valid after read cycle
                        BTL_Read_Request = 0;                            // no more cycles for now
                else {                                                   // LMU mode
                        Y0_FIFO_ACK = 1;                                 // read BTL memory to Y0
                        LMU_Read_Type = 1;
                        BTL_Read_Request = 1;                            // request cycle for read of Y263
                }
        }
        else if ( LMU_Read_Type == 1 ) {
                Y263_FIFO_ACK = 1;                                       // read Y263 from LMU memory 0
                Memory_Address = LMU_Read_Address;
                LMU_Read_Address = LMU_Read_Address + 1;                 // point to LMU memory 1
                LMU_Read_Type = 2;
                BTL_Read_Request = 1;                                    // request cycle for read of M263
        }
        else if ( LMU_Read_Type == 2 ) {
                M263_FIFO_ACK = 1;                                       // read M263 from LMU memory 1
                Memory_Address = LMU_Read_Address;
                LMU_Read_Address = LMU_Read_Address + 1;                 // point to next word of LMU mem 0
                LMU_Read_Type = 0;
                BTL_Read_Request = 0;                                    // No more cycles needed now
                LMU_Read_Word_Count = LMU_Read_Word_Count + 1;           // count words
                if ( LMU_Read_Word_Count == BlockPairs_per_Line ) {      // see if end of line
                        LMU_Read_Address = LMU_Line0_Start * 4;          // yes, reset LMU pointer
                        LMU_Read_Word_Count = 0;
                }
        }
}
                                                                 //Otherwise, see if read cycle request is needed
else if ( BTL_Data_Valid == 1 ) {
        if ( Number_Output_Line_Mem == 0 ) {
                if ( Output_Read_FIFO_REQ(0) = 1 ) BTL_Read_Request = 1;
        }
        else if ( LMU_Mode == 0 ) {
                if ( FB_Latch_Valid == 0 ) BTL_Read_Request = 1;
        }
        else if ( Y0_FIFO_REQ == 1 ) BTL_Read_Request = 1;
}
```

5.3.2.3. Write to Output and LMU Line Memories

When not in LMU mode this process simply writes a 128 bit word of data to one of the N output line memories; data is obtained from the feedback latch, which is loaded by the block-to-line read process. The FB_Latch_Valid flag indicates when valid data is available in the latch.

When operating in LMU mode, for every 16 pixels processed a word must be written to two output line memories and two LMU line memories; all four words become available from the LMU section simultaneously. This process issues a memory access request when the Y262 data is available; after writing the first word, three more requests and writes are performed to complete the needed data transfer. Data comes from either the 8 bit to 128 bit FIFOs (which use the RDY/ACK protocol on their output side) or the feedback latch. When writing to the output line memories, the Y262 or Y_EST FIFOs are the sources; Y262 is simultaneously written into the feedback latch. When writing to the LMU line memories, data is taken from the feedback latch or the M262 FIFO.

This process is reset by V_RST. It is synchronized with the read of the output line memories by means of a Output_Write_Limit signal from the output read process, which conveys the last address read from a particular output line memory (as specified by the VFC). Writing by this process is inhibited when the output memory write pointer matches that address. By selecting the line memory to be write protected, the VFC can control the data which output memories contain at the time it reads them; this system assumes that the write of the output memories can proceed at a rate faster than the read rate. An Output_Line_Data_Valid flag indicates to the read process that the output memories have been fully written; at the start of each field writing is constantly enabled until the flag is set.

The following resources are used by this process:

Context Registers: Output_Write_Line0_Start(9:2) // start address of output line memories

LMU_Line0_Start(9:2) // start address of LMU line memories

External Interfaces: BlockPairs_per_Line(6:0) // equal to picture width in macroblocks

Number_Output_Line_Mem(2:0) // number of memories used (0 to 4)

Output_Write_Limit(9:0) // address limit for write process

Output_Write_Enable // grants access to memory (from arbitrator)

Y262_FIFO_RDY // ready line from the Y262 FIFO

Y262_FIFO_ACK // acknowledge to the Y262 FIFO

Y_EST_FIFO_ACK // acknowledge to the Y_EST FIFO

M262_FIFO_ACK // acknowledge to the M262 FIFO

Mux1_Control // selects the FB Latch out as memory input source

Mux2_Control // selects the memory input as FB Latch input

FB_Latch_Enable // enable the feedback latch to latch data

LMU_Mode // indicates that LMU mode is in effect

Memory_Address(9:0) // Address bus to display RAM

FB_Latch_Valid // flag to indicate that the latch contents are valid

Working Registers: Output_Write_Address(9:0) // present output write address

LMU_Write_Address(9:0) // present LMU line memory write address

Output_Write_Word_Count(6:0) // words written in present line

Output_Write_Line_Count(2:0) // lines written

Output_Write_Request // request to arbitrator for access to memory

Output_Line_Data_Valid // flag that the line memories have data

LMU_Write_Type(1:0) // specifies type of LMU data transfer

Process Description (executes every potential memory cycle)

```
Mux1_Control = Mux2_Control = 0;                                    // default conditions
Y_EST_FIFO_ACK = Y262_FIFO_ACK = M262_FIFO_ACK = FB_Latch_Enable = 0;
if ( V_RST == 1 ) {                                                        // Initialize
        Output_Line_Data_Valid = 0;
        Output_Write_Address = Output_Write_Line0_Start * 4;
        LMU_Write_Address = LMU_Line0_Start * 4;
        Output_Write_Word_Count = Output_Write_Line_Count = 0;
        LMU_Write_Type = 0;
}
else if ( Output_Write_Enable == 1 ) {              // ready to write a word to a line memory
        Memory_WEB = 0;                                      // enable memory for writing
        if ( LMU_Mode == 0 ) {                               // write latch to output memory
                Mux1_Control = 1;
                FB_Latch_Valid = 0;
                Memory_Address = Output_Write_Address;
                Output_Write_Address = Output_Write_Address + 1;
                Output_Write_Request = 0;
        }
        else {                                                          // LMU mode, so check where we are
            if ( LMU_Write_Type == 0 ) {                                   // write Y262 to output memory
                Y262_FIFO_ACK = 1;
                Mux2_Control = 1;
                FB_Latch_Enable = 1;
                Memory_Address = Output_Write_Address;
                Output_Write_Address = Output_Write_Address + 1;
                Output_Write_Word_Count = Output_Write_Word_Count + 1;
                LMU_Write_Type = 1;
                Output_Write_Request = 1;
            }
            else if ( LMU_Write_Type == 1 ) {                           // write Y_EST to output memory
                Y_EST_FIFO_ACK = 1;
                Memory_Address = Output_Write_Address;
                Output_Write_Address = Output_Write_Address + 1;
                Output_Write_Word_Count = Output_Write_Word_Count + 1;
                LMU_Write_Type = 2;
                Output_Write_Request = 1;
            }
            else if ( LMU_Write_Type == 2 ) {                              // write Y262 to LMU memory
                Mux1_Control = 1;
                Memory_Address = LMU_Write_Address;
                LMU_Write_Address = LMU_Write_Address + 1;
                LMU_Write_Type = 3;
                Output_Write_Request = 1;
            }
            else if ( LMU_Write_Type == 3 ) {                              // write M262 to LMU memory
                M262_FIFO_ACK = 1;
                Memory_Address = LMU_Write_Address;
                LMU_Write_Address = LMU_Write_Address + 1;
                LMU_Write_Type = 0;
                Output_Write_Request = 0;
            }
        }
        if ( Output_Write_Word_Count == BlockPairs_per_Line ) {           // see if at end of output memory
            Output_Write_Word_Count = 0;
            Output_Write_Line_Count = Output_Write_Line_Count + 1;
            if ( Output_Write_Line_Count = Number_Output_Line_Mem ) {
                Output_Line_Data_Valid = 1;
                Output_Write_Line_Count = 0;
            }
        }
}
                                        // Otherwise, if ready then request access for a write cycle
else if ( ( Output_Line_Data_Valid == 0 || Output_Write_Address != Output_Write_Limit ) &&
        ( ( LMU_Mode == 0 && FB_Latch_Valid == 1 ) || ( LMU_Mode == 1 && Y262_FIFO_RDY == 1 ) ) )
        Output_Write_Request = 1;
}
```

C5.3.2.4. Read from Output Line Memories

N output line memories are used to provide signals for the vertical format converter (VFC) function, where N can be between zero and four depending on the display mode. The signals represent vertically aligned adjacent pixels of the source picture. The line memories are allocated as a contiguous block of display memory. During a given display line period, all or some subset of these memories need to be read, as specified by the VFC. Starting with the lowest numbered active line memory, a word is read from each active line memory in sequence; in subsequent executions the corresponding word from the remaining active memories is read until all have been read. The address pointer is then repositioned to the next word of the lowest numbered active line memory, and the process repeats. No checking is done for end of line.

The data is sent to the VFC via four 128 to 8 bit conversion FIFOs. The input side of the FIFOs use a REQ/ACK protocol. The REQ line from the FIFO corresponding to the next line memory to be read is used as a request for this process. The output side of the FIFOs use a RDY/ACK protocol; the VFC will AND together the RDY lines from the active FIFOs to determine when data is available at the start of each display line period. The FIFOs and this process are reset by H_RST, so the line memory contents are re-read every line starting at an address specified by the user bus register Output_Read_Line0_Start; by setting a value into this register which is larger than the Output_Write_Line0_Start value, a pan function (in units of 8 luma pixels) can be performed.

The writing of the output line memories is synchronized with the reading using an Output_Write_Limit register. A Output_Line_Protect signal from the VFC specifies a particular line memory to protect; whenever this line memory is read, the Output_Write_Limit register is loaded with the read address being used. The write process is inhibited whenever the write address matches this value.

The following resources are used by this process:

Context Registers: Output_Read_Line0_Start(9:0) // start address of line memory storage

External Interfaces: BlockPairs_per_Line(6:0) // equal to picture width in macroblocks

Number_Output_Line_Mem(2:0) // number of memories used (from VFC)

Output_Line_Data_Valid // flag from write process that all mem written

Output_Line_Enab(3:0) // active memories to be read (4 flags from VFC)

Output_Line_Protect(2:0) // line memory to protect (from VFC)

Output_Read_Enable // access to memory granted (flag from arbitrator)

Output_Read_FIFO_REQ(3:0) // request lines from the output FIFOs

Memory_Address(9:0) // Address bus to display RAM

Working Registers: Output0_Read_Address(9:0) // line 0 read address

Output_Read_Address(9:0) // present read address

Output_Read_Line(2:0) // number of line memory currently being read

Output_Write_Limit(9:0) // address limit used to control write process

Output_Read_FIFO_ACK(3:0) // ACKs to FIFOs

Output_Read_Done // flag indicating the address needs to be updated

Output_Read_Request // request to arbitrator for access to memory

<u>Process Description</u> (executes every potential memory cycle)

```
for( i = 0; i < 4; ++i ) Output_Read_FIFO_ACK (i) = 0;                    // default conditions
if ( H_RST == 1 ) {                                                         // initialize
```

```
        Output_Read_Line = Output_Read_Done = 0;
        Output_Read_Address = Output0_Read_Address = Output_Read_Line0_Start;
}
else if ( Output_Read_Enable == 1 ) {                              // read a word from the line memory
        Output_Read_Request = 0;
        Memory_Address = Output_Read_Address;
        Memory_WEB = 1;
        Output_Read_Done = 1;
        Output_Read_FIFO_ACK ( Output_Read_Line ) = 1;          // Enable FIFO (must delay signal 1 cycle)
        if ( Output_Line_Protect == Output_Read_Line ) Output_Write_Limit = Output_Read_Address;
}
                                                          // Calculate new read address value if needed
else if ( Output_Read_Done == 1 || Output_Line_Enab( Output_Read_Line ) == 0 ) {
        Output_Read_Done = 0;
        Output_Read_Address = Output_Read_Address + BlockPairs_per_Line;
        Output_Read_Line = Output_Read_Line + 1;
        if ( Output_Read_Line == Number_Output_Line_Mem ) {
                Output_Read_Line = 0;
                Output0_Read_Address = Output0_Read_Address + 1;
                Output_Read_Address = Output0_Read_Address;
        }
}
                            // everything set, so if FIFO needs data and it's available request a memory cycle
else if ( Output_Line_Data_Valid == 1 && Output_Read_FIFO_REQ ( Output_Read_Line ) == 1 )
        Output_Read_Request = 1;
```

C5.3.2.5. Refresh

This process refreshes the dynamic memory. It cycles through the entire 10 bit address range reading memory; it is assumed that there is no problem attempting to read non-existent addresses.

The following resources are used by this process:

Context Registers: Display_Refresh_Interval(7:0) // memory cycles per refresh cycle

Display_Refresh_Increment(7:0) // address increment

External Interfaces: Display_Refresh_Enable // access to memory granted (flag from arbitrator)

Memory_Address(9:0) // Address bus to display RAM

Working Registers: Display_Refresh_Address(9:0) // present refresh address

Display_Refresh_Count(7:0) // memory cycles to next refresh

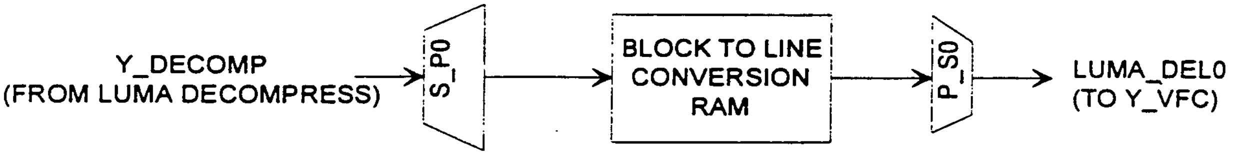
Display_Refresh_Request // request to arbitrator for access to memory

<u>Process Description</u> (executes every potential memory cycle)

```
if ( V_RST == 1 ) {                                                    // initialize on vertical reset
        Display_Refresh_Request = 0;
        Display_Refresh_Address = 0;
        Display_Refresh_Count = Display_Refresh_Interval;
}
else if ( Display_Refresh_Enable == 1 ) {
        Display_Refresh_Request = 0;
        Memory_Address = Display_Refresh_Address;
        Memory_WEB = 1;                                         // use read cycle for refresh
        Display_Refresh_Address = Display_Refresh_Address + Display_Refresh_Increment;
}
else {
        Display_Refresh_Count = Display_Refresh_Count - 1;
         if ( Display_Refresh_Count == 0 ) {
                Display_Refresh_Count = Display_Refresh_Interval;
                Display_Refresh_Request = 1;
        }
}
```

C5.3.3. Luma RAM functional modes

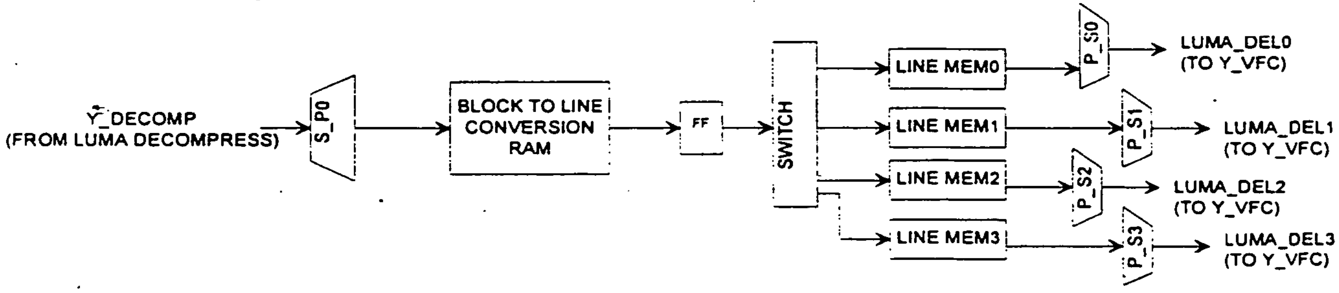
In each different major mode of operation (block-to-line conversion only, adding one to four luma line memories, LMU mode) the RAM has a different memory mapping and a different data flow. These changes are enabled by the two feedback muxes (MUX1, MUX2), and the enabled flip-flop in between. The following diagrams show a function, data flow, view of the memory for each mode.



YBTLRAM3.DS4
12/08/95

FIGURE C 6. Block to Line only Mode

This is the simplest mode - "full HD", where the entire RAM is used for block-to-line conversion, and there is no vertical filtering.



YBTLRAM2.DS4
12/08/95

FIGURE C 7. Block to Line and Vertical Format Conversion Mode

In this mode, part of the RAM is used for block-to-line conversion. Then output of the block-to-line conversion is written alternately to up to four line memory spaces. After a line delay, up to four of the line memories are read. Care must be taken when writing to the line memory being written.

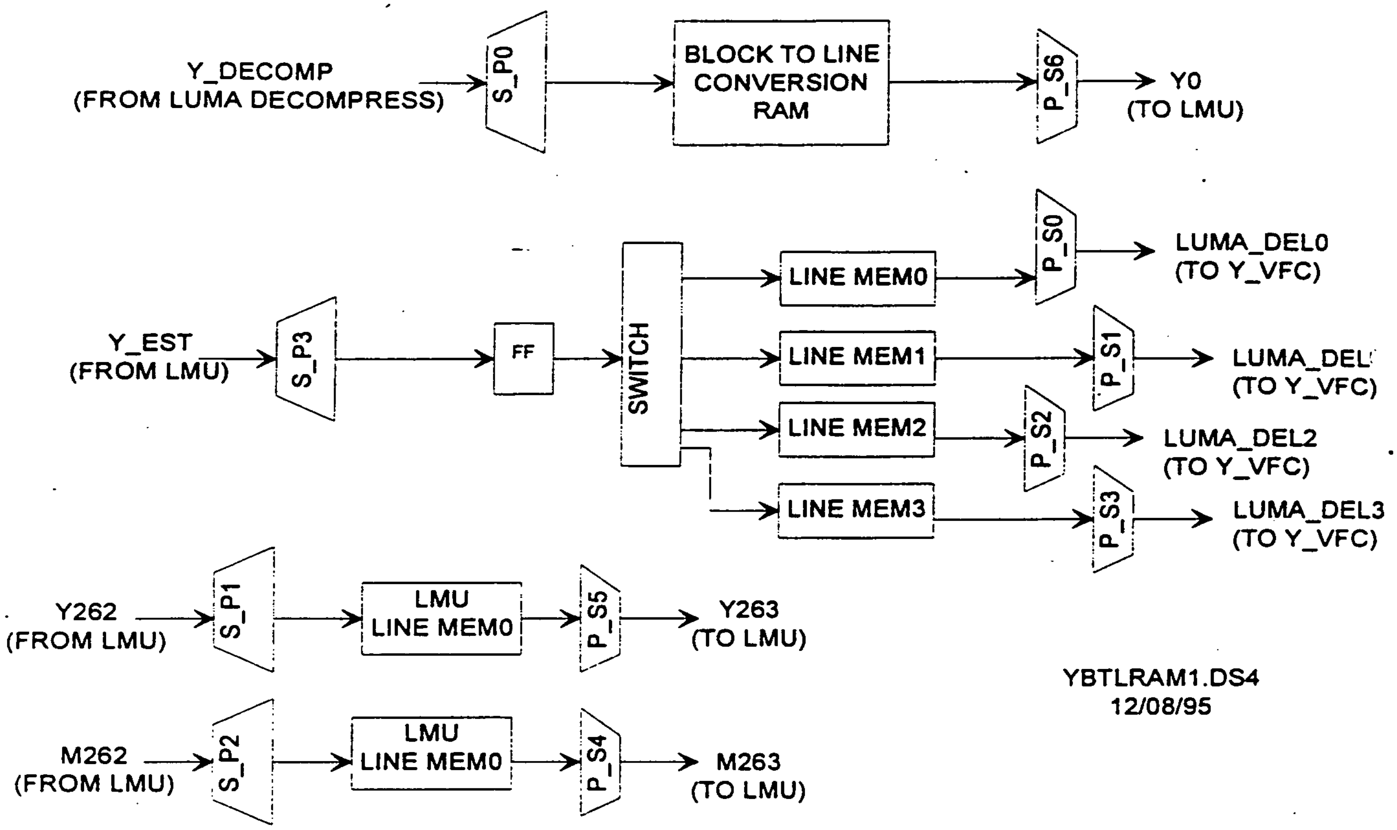
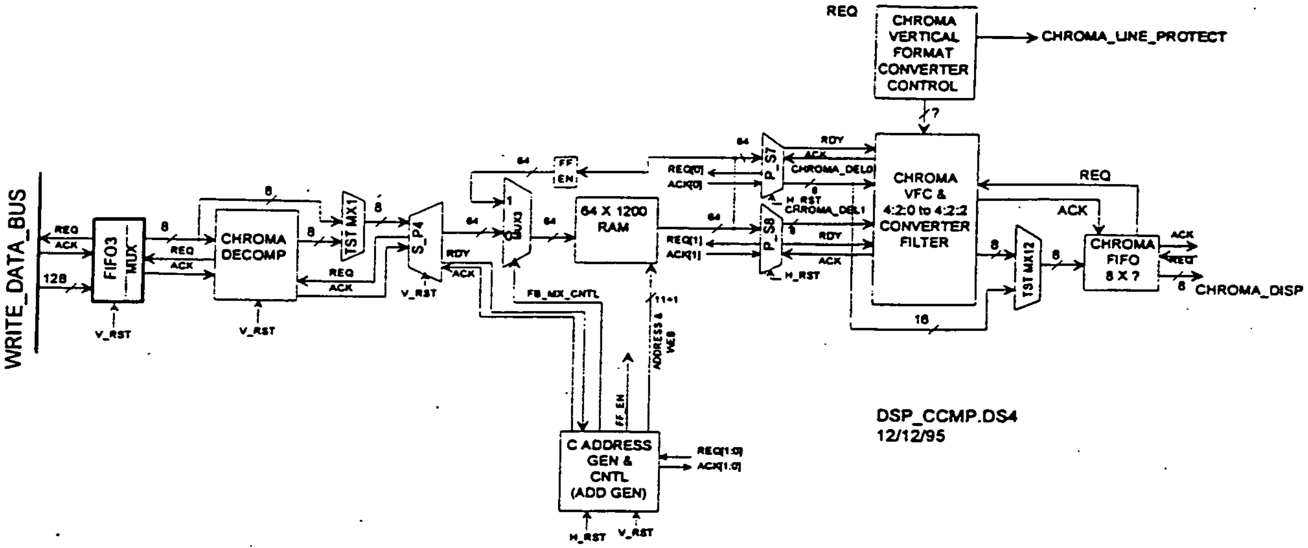


FIGURE C 8. Block to Line, Vertical Format Conversion, and LMU Mode

This is by far the most complicated mode. Here, the output of the block-to-line conversion is sent to the LMU block, and one of the LMU block outputs (Y_EST) is written alternately to up to four line memory spaces (as in the previous mode). After a line delay, up to four of the line memories are read. Care must be taken when writing to the line memory being written. In addition, two LMU block outputs are written into two seperate line memory spaces to be read back out later and passed back to the LMU.

C5.3.4. Chroma Decompress and Block to Line Conversion



DSP_CCMP.DS4
12/12/95

FIGURE C 9. DISPLAY CHROMA DECOMPRESS, BLOCK CONVERSION RAM and VERTICAL FILTER

As shown in Figure C9, the chroma channel has a RAM used for block-to-line conversion and line memory. The memory compression technique encodes the chroma in blocks of 4 by 4 pixels, for each chroma component. The decompress block interleaves the two 4x4 blocks, and the data passed out of the chroma display decompress block is a sequence of pixels from two interleaved 4x4 blocks. After the serial-to-parallel converter (S-P4), one 64 bit-wide (4 pixels Cr, 4 pixels Cb,interleaved) word is ready for a write to the RAM.

Unlike the luma block to line conversion RAM system, the chroma system has only one operating mode. The block to line conversion space uses 960 of the 1200 RAM addresses. The data is read from that space, and using the feed-back flip-flop and MUX, the data is written to the remaining space reserved for two lines of chroma memory. The data is read from the line memory area of RAM and is sent to the two parallel-to-serial converters (P_S7, P_S8).

The outputs of P_S7, P_S8 are the two chroma signals which become the inputs to the chroma vertical format converter (C - VFC), which also does 4:2:0 to 4:2:2 conversion. The output of the C-VFC is passed onto a FIFO which acts as a buffer between the VFC (in the decompress clock domain) and the horizontal sample rate converter (H-SRC) which is in the display clock domain. The usual REQ/ACK handshaking is used to regulate data going into and out of the FIFO.

The S_P and P_S FIFO's use the same control structures as those for luma.

C5.3.5. Chroma Ram Address Generator (C_ADD_GEN)

The chroma address generator is essentially to that shown in Figure C5 for the luma address generator. The only real difference is that the address space requires an 11-bit address, and the individual processes (Write block to line memory, Read block to line memory, Write output memory, Read output memory, refresh) are simpler than those for luma. This is because there is only one mode, there is no LMU, and there are only two (not 4) line memories to be controlled. The function of the arbitrator is the same.

The chrominance display memory is composed of 1200 words of 64 bits each. The first 960 words are used as the block-to-line conversion memory; the remaining 240 words form two output line memories. Processes that access the block-to-line portion generate a 10 bit address, to which a msb zero is appended to produce an 11 bit address to the memory. Processes that access the output line memories generate an 8 bit address, to which the number 960 is added to produce an 11 bit address for the memory.

C5.3.5.1. Write to Block-to-Line Conversion Memory

This process accepts a 64 bit word from the FIFO connected to the output of the memory decompress block. The word corresponds to alternate 8 bit C_R and C_B values of 4 horizontally adjacent pixels from a C_R and C_B pair of 4 x 4 blocks of pixels. The word is written to the block-to-line conversion portion (bottom) of memory, which stores one complete row of 8 x 4 block pairs. Address calculations make use of a parameter called CBTL_Modulus, which has the value (4 x BlockPairs_per_Line - 1); this is numerically equal to the value of the highest address used.

This process is reset by V_RST, which also resets a CBTL_Data_Valid flag. At the beginning of each row of blocks, the write address is set to zero; after each word is written, an increment value (given by CBTL_delta) is added; the result is then reduced modulo CBTL_Modulus. This algorithm would result in a value of zero for the last word of the block row; when this is detected a CBTL_Write_Last flag is set, and word is stored at an address equal to CBTL_Modulus instead. The memory is completely filled once at the start of each field, at which point the CBTL_Data_Valid flag is set; after that, a word is not written until the data previously stored at that location has been read by the block to line read process. This is accomplished by inhibiting this write process when the write address matches the address being used by the read process.

The value of CBTL_delta changes for each row of blocks processed. The first row of blocks in a field is written using a value of one. For each subsequent row, a new value of CBTL_delta is calculated by multiplying the previous value by 4 and reducing the result modulo CBTL_Modulus; this same value is used when calculating addresses for both reading the previously stored data and writing the new data.

The following resources are used by this process:

Context Registers: none

External Interfaces: BlockPairs_per_Line(6:0) // equal to picture width in macroblocks

CBTL_Modulus(9:0) // equal to 4 * BlockPairs_per_line - 1

CBTL_Read_Address(9:0) // present block-to-line read address

CBTL_Write_Enable // grants access to memory (from arbitrator)

CDCD_FIFO_RDY // ready line from the chroma DeCompress Data FIFO

CDCD_FIFO_ACK // acknowledge to the chroma DeCompress Data FIFO

CMemory_Address(10:0) // Address bus to chroma display RAM

Working Registers: CBTL_Write_Address(9:0) // present block-to-line write address

CBTL_delta(9:0) // conversion delta value for this row of blocks

CBTL_delta_next(11:0) // delta value for the next row of blocks

CBTL_Write_Last // ready to write last word of a block row

CBTL_Data_Valid // flag to indicate that the BTL memory is full

CBTL_Write_Request // request to arbitrator for access to memory

Temporary Variables: Temp1(10:0), Temp2(10:0), Temp3(11:0) // used in address calculation

Process Description (executes every potential memory cycle)

```
CDCD_FIFO_ACK = 0;                                                          // default conditions
if ( V_RST == 1 ) {                                                   // Initialize on vertical reset
        CBTL_Write_Address = CBTL_Data_Valid = CBTL_Write_Last = CBTL_Write_Request = 0;
        CBTL_delta = 1;
        CBTL_delta_next = 4;
}
else if ( CBTL_Write_Enable == 1 ) {                            // access to write a word has been granted
        CBTL_Write_Request = 0;                                                        // disable request
        CMemory_WEB = 0;                                                  // enable memory for writing
        CMemory_Address = CBTL_Write_Address;     // extend address from 10 to 11 bits by appending msb 0
        CDCD_FIFO_ACK = 1;                                            // enable CDCD FIFO to output data
        Temp1 = CBTL_Write_Address + CBTL_delta;                   // calculate next BTL write address
        Temp2 = Temp1 - CBTL_Modulus;                                                  // subtract modulus
        if ( CBTL_Write_Last == 1 ) {                                 // check if writing last word of row
                CBTL_Write_Last= 0;
                CBTL_Write_Address = 0;                                         // reset pointer to start
                CBTL_Data_Valid = 1;                          // indicate that the memory has been filled
                CBTL_delta = CBTL_delta_next;                                       // update delta value
                CBTL_delta_next = 4 * CBTL_delta_next;          // calculate unreduced next delta value
        }
        else if ( Temp2 == 0 ) {                                    // address calculation result is zero
                CBTL_Write_Last = 1;                         // this will be the last write for the block row
                CBTL_Write_Address = Temp1;                                         // set to CBTL_Modulus
        }
        else if ( Temp2 < 0 ) CBTL_Write_Address = Temp1;         // subtraction result negative, don't use
        else CBTL_Write_Address = Temp2;                                          // use subtraction result
}
                                                   // Otherwise, see if write cycle needs to be requested
else if ( ( CBTL_Data_Valid == 0 || CBTL_Write_Address != CBTL_Read_Address ) && CDCD_FIFO_RDY == 1 )
        CBTL_Write_Request = 1;
                                                       // Otherwise, reduce next delta value if necessary
else {
        temp3 = CBTL_delta_next - CBTL_Modulus;
        if ( temp3 >= 0 ) CBTL_delta_next = temp3;
}
```

C5.3.5.2. Read from Block-to-Line Conversion Memory

This process reads a 64 bit word from the block-to-line conversion portion (bottom) of chroma memory and stores it in the feedback latch. The new read address is then calculated using a CBTL_delta value obtained from the block-to-line write process.

This process is reset by V_RST. It is synchronized with the write of the block-to-line conversion by preventing the write process from proceeding whenever the write address is equal to the read address generated by this process. A CBTL_Data_Valid flag indicates that the block-to-line memory has been fully written at least once; at the start of each field, reading is inhibited until this flag is high.

The following resources are used by this process:

Context Registers: None

External Interfaces: BlockPairs_per_Line(6:0) // equal to picture width in macroblocks

CBTL_Write_Address(9:0) // present block-to-line write address (from CBTL write)

CBTL_Read_Enable // grants access to memory (from arbitrator)

CFB_Latch_Enable // enable the feedback latch to latch data

CBTL_delta(9:0) // conversion delta value for this row of blocks (from CBTL write)

CBTL_Data_Valid // flag to indicate that the BTL memory is full (from CBTL write)

CBTL_Modulus(9:0) // equal to 4 * BlockPairs_per_line - 1 (from CBTL write)

CMemory_Address(10:0) // Address bus to chroma display RAM

Working Registers: CBTL_Read_Address(9:0) // present block-to-line read address

CBTL_Read_Last // ready to read last word of a block row

CFB_Latch_Valid // indicates that the latch contents are valid

CBTL_Read_Request // request to arbitrator for access to memory

Temporary Variables: Temp1(10:0), Temp2(10:0) // used in address calculation

Process Description (executes every potential memory cycle)

```
CFB_Latch_Enable = 0;                                   // default conditions
if ( V_RST == 1 ) {                                     // Initialize on vertical reset
        CBTL_Read_Address = 0;
        CFB_Latch_Valid = 0;
        CBTL_Read_Last = 0;
        CBTL_Read_Request = 0;
}
else if ( CBTL_Read_Enable == 1 ) {                     // access to read a word has been granted
        CMemory_WEB = 1;                                // enable memory for reading
        CMemory_Address = CBTL_Read_Address;    // extend address from 10 to 11 bits by appending msb 0
        Temp1 = CBTL_Read_Address + CBTL_delta;         // calculate next CBTL read address
        Temp2 = Temp1 - CBTL_Modulus;
        if ( CBTL_Read_Last == 1 ) {                    // start new block row
                CBTL_Read_Last = 0;                     // reset pointer to start
                CBTL_Read_Address = 0;
        }
        else if ( Temp2 == 0 ) {
                CBTL_Read_Last = 1;                     // this will be the last read for the block row
                CBTL_Read_Address = Temp1;
        }
        else if ( Temp2 < 0 ) CBTL_Read_Address = Temp1;
        else CBTL_Read_Address = Temp2;
        CFB_Latch_Enable = 1;                           // enable latch to capture data
        CFB_Latch_Valid = 1;                            // latch data will be valid after read cycle
        CBTL_Read_Request = 0;                          // disable requests
}
                                                        //Otherwise, see if read cycle request is needed
else if ( CBTL_Data_Valid == 1 && CFB_Latch_Valid == 0 ) CBTL_Read_Request = 1;
```

C5.3.5.3.Write to Output Line Memories

This process writes a 64 bit word of data to one of the 2 output line memories; data is obtained from the feedback latch, which is loaded by the block-to-line read process. The CFB_Latch_Valid flag indicates when valid data is available in the latch.

This process is reset by V_RST. It is synchronized with the read of the output line memories by means of a COutput_Write_Limit signal from the output read process, which conveys the last address read from one of the two output line memories (as specified by the VFC). Writing by this process is inhibited when the output memory write pointer matches that address. By selecting the line memory to be write protected, the VFC can control the data which output memories contain at the time it reads them. A COutput_Line_Data_Valid flag indicates to the read process that the output memories have been fully written; at the start of each field writing is constantly enabled until the flag has been set.

The following resources are used by this process:

Context Registers: none

External Interfaces: BlockPairs_per_Line(6:0) // equal to picture width in macroblocks

COutput_Write_Limit(7:0) // address limit for write process

COutput_Write_Enable // grants access to memory (from arbitrator)

CFB_Latch_Enable // enable the feedback latch to latch data

CMux1_Control // selects the FB Latch out as memory input source

CMemory_Address(10:0) // Address bus to chroma display RAM

CFB_Latch_Valid // flag indicating that the latch contents are valid

Working Registers: COutput_Write_Address(7:0) // present output write address

COutput_Write_Request // request to arbitrator for access to memory

COutput_Line_Data_Valid // flag that the line memories have data

Process Description (executes every potential memory cycle)

```
CFB_Latch_Enable = CMux1_Control = 0;
if ( V_RST == 1 ) {                                                          // Initialize
        COutput_Line_Data_Valid = COutput_Write_Address = 0;
}
else if ( COutput_Write_Enable == 1 ) {                   // ready to write a word to a line memory
        CMemory_WEB = 0;                                             // enable memory for writing
        CMux1_Control = 1;                                    // select latch as memory data source
        CFB_Latch_Valid = 0;                                  // show that latch data has been used
        CMemory_Address = COutput_Write_Address + 960;  // extend address from 8 to 11 bits by adding 960
        COutput_Write_Address = COutput_Write_Address + 1;
        COutput_Write_Request = 0;
        if ( COutput_Write_Address == 2 * BlockPairs_per_Line ) {          // see if output memory is full
                COutput_Write_Address = 0;
                COutput_Line_Data_Valid = 1;
        }
}
```

```
                                                    // Otherwise, if ready then request access for a write cycle
else if ( ( COutput_Line_Data_Valid == 0 || COutput_Write_Address != COutput_Write_Limit )
                                                                               && CFB_Latch_Valid == 1 )
            COutput_Write_Request = 1;
}
```

C5.3.5.4.Read from Output Line Memories

Two output line memories are used to provide chroma signals for the vertical format converter (VFC) function. The signals represent vertically aligned adjacent pixels of the source picture. The data is sent to the VFC via two 64 to 8 bit conversion FIFOs, which use a REQ/ACK protocol on their inputs. The REQ line from the FIFO corresponding to line memory 0 is used as a request for this process. The output side of the FIFOs use a RDY/ACK protocol; the VFC uses the RDY line from the FIFO for line memory 1 to determine when data is available at the start of each display line period. The FIFOs and this process are reset by H_RST, so the line memory contents are re-read every line, starting from an offset address specified by the user bus register COutput_Read_Start_Address; by setting a non-zero value into this register, a pan function (in units of 4 chroma pixels) can be performed.

The writing of the output line memories is synchronized with the reading using an COutput_Write_Limit register. A COutput_Line_Protect signal from the VFC specifies a particular line memory to protect; whenever this line memory is read, the COutput_Write_Limit register is loaded with the read address being used. The write process is inhibited whenever the write address matches this value.

The following resources are used by this process:

Context Registers: COutput_Read_Start_Address(7:0) // start address of line memory read

External Interfaces: BlockPairs_per_Line(6:0) // equal to picture width in macroblocks

COutput_Line_Data_Valid // flag from write process that all mem written

COutput_Line_Protect // chroma line memory to protect (from VFC)

COutput_Read_Enable // access to memory granted (flag from arbitrator)

COutput_Read_FIFO_REQ(1:0) // request lines from the output FIFOs

CMemory_Address(10:0) // Address bus to chroma display RAM

Working Registers: COutput_Read_Address(7:0) // present read address

COutput_Read_LineNo // number of line memory currently being read

COutput_Write_Limit(7:0) // address limit used tc control write process

COutput_Read_FIFO_ACK(1:0) // ACKs to FIFOs

COutput_Read_Request // request to arbitrator for access to memory

<u>Process Description</u> (executes every potential memory cycle)

```
COutput_Read_FIFO_ACK (0) = COutput_Read_FIFO_ACK (1) = 0;                    // default conditions
if ( H_RST == 1 ) {                                                                    // initialize
        COutput_Read_LineNo = 0;
        COutput_Read_Address = COutput_Read_Start_Address;
}
else if ( COutput_Read_Enable == 1 ) {                                 // read a word from the line memory
        CMemory_Address = COutput_Read_Address + 960;   // extend address from 8 to 11 bits by adding 960
        CMemory_WEB = 1;
        COutput_Read_FIFO_ACK ( COutput_Read_LineNo ) = 1;                                   // Enable FIFO
        if ( COutput_Line_Protect == COutput_Read_LineNo ) COutput_Write_Limit = COutput_Read_Address;
        COutput_Read_LineNo = ! COutput_Read_LineNo;
        if ( COutput_Read_LineNo == 1 ) {
                COutput_Read_Address = COutput_Read_Address + BlockPairs_per_Line;        // point to line 1
                COutput_Read_Request = 1;                                         // request cycle to read line 1
        }
        else {                                                                  // point to the next word of line 0
                COutput_Read_Address = COutput_Read_Address + 1 - BlockPairs_per_Line;
                COutput_Read_Request = 0;                                               // no immediate request
}
                                            // if FIFO needs data and it's available request a memory cycle
else if ( COutput_Line_Data_Valid == 1 && COutput_Read_FIFO_REQ(0) == 1 )
        COutput_Read_Request = 1;
```

C5.3.5.5. Refresh

This process refreshes the dynamic memory. It cycles through the entire 11 bit address range reading memory; it is assumed that there is no problem attempting to read non-existent addresses.

The following resources are used by this process:

Context Registers: none

External Interfaces: CDisplay_Refresh_Enable // access to memory granted (flag from arbitrator)

CMemory_Address(10:0) // Address bus to chroma display RAM

Display_Refresh_Interval(7:0) // memory cycles per refresh cycle (from luma)

Display_Refresh_Increment(7:0) // address increment (from luma)

Working Registers: CDisplay_Refresh_Address(10:0) // present refresh address

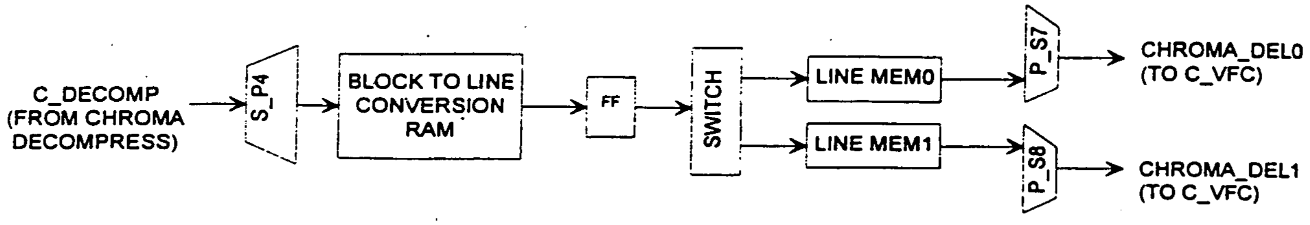
CDisplay_Refresh_Count(7:0) // memory cycles to next refresh

CDisplay_Refresh_Request // request to arbitrator for access to memory

<u>Process Description</u> (executes every potential memory cycle)

```
if ( V_RST == 1 ) {                                                        // initialize on vertical reset
        CDisplay_Refresh_Request = 0;
        CDisplay_Refresh_Address = 0;
        CDisplay_Refresh_Count = Display_Refresh_Interval;
}
else if ( CDisplay_Refresh_Enable == 1 ) {
        CDisplay_Refresh_Request = 0;
        CMemory_Address = CDisplay_Refresh_Address;// extend address from 10 to 11 bits by appending msb 0
        CMemory_WEB = 1;
        CDisplay_Refresh_Address = CDisplay_Refresh_Address + Display_Refresh_Increment;
}
else {
        CDisplay_Refresh_Count = CDisplay_Refresh_Count - 1;
        if ( CDisplay_Refresh_Count == 0 ) {
                CDisplay_Refresh_Count = Display_Refresh_Interval;
                CDisplay_Refresh_Request = 1;
        }
}
```

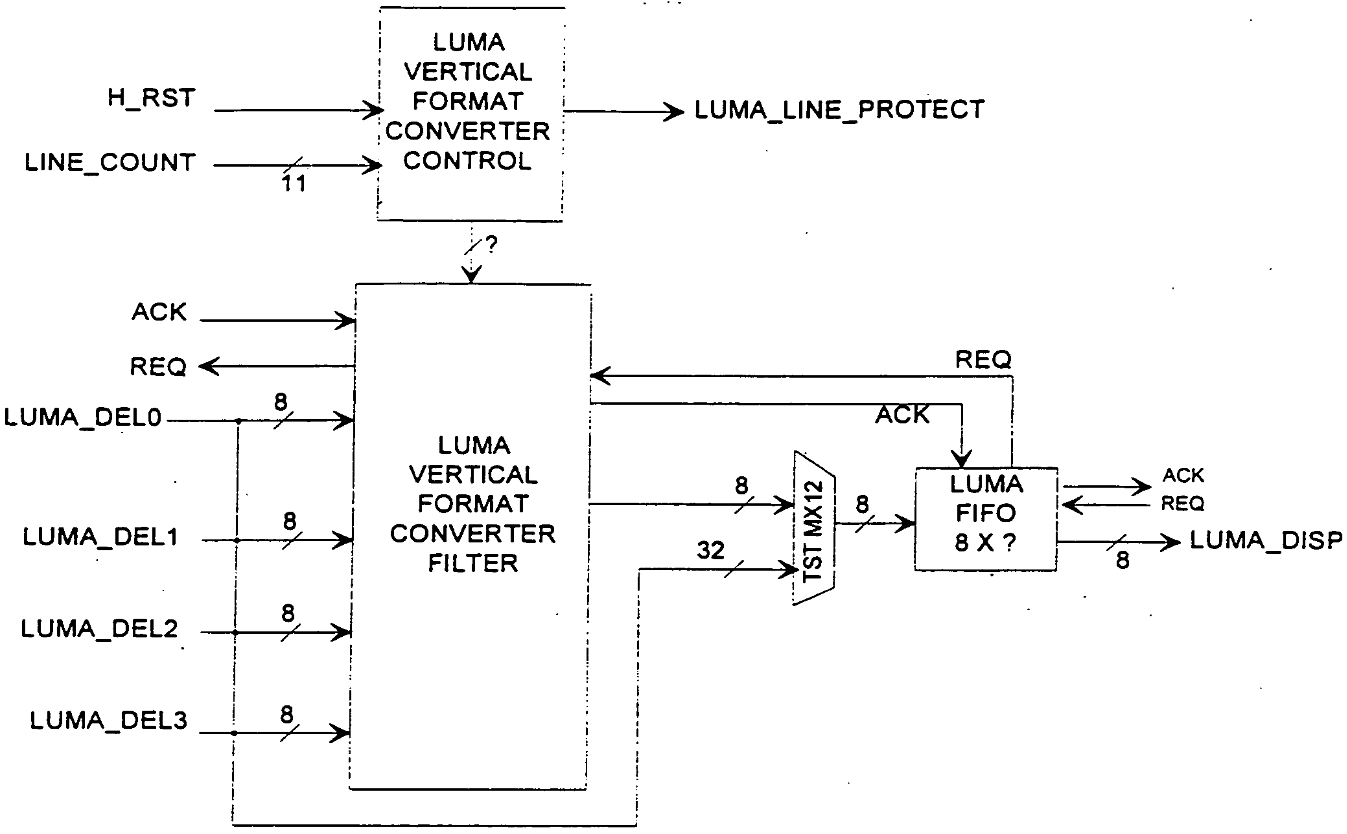
C5.3.6. Chroma RAM functional modes



CBTLRAM.DS4
12/12/95

FIGURE C 10. Block to Line and Vertical Format Conversion Mode

A functional diagram showing the data flow for the chroma block to line RAM and line memory system is shown in Figure C10.



DSPYCMP2.DS4
11/21/95

FIGURE C 9. DISPLAY LUMA VERTICAL FORMAT CONVERTER TOP LEVEL

The top level view of the Luma Vertical Format Converter (Y VFC) is shown in Figure C 9 below. The signal inputs are the outputs of four luma line delays (LUMA_DEL0 .. LUMA_DEL3). The

C5.4. Vertical Sample Rate Converter

In some modes, the three inputs (luma primary, luma secondary, luma delayed) are read directly from the external memory. In other modes, the luma feedback path is used to perform some of the vertical filter calculations. The vertical filter output is calculated as needed by the H-SRC (horizontal sample rate converter) to eliminate the need for large FIFOs between the vertical and horizontal sample rate converters.

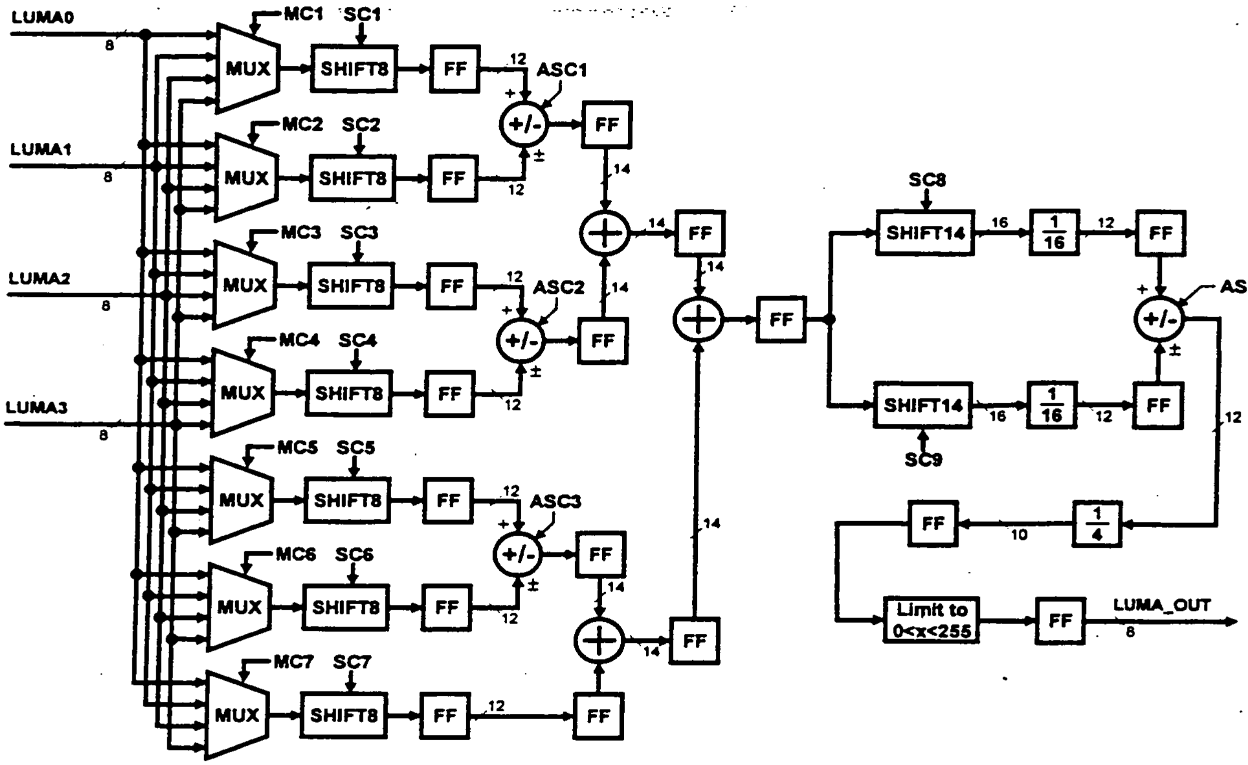


Figure C 3 -- Luminance Vertical Format Converter

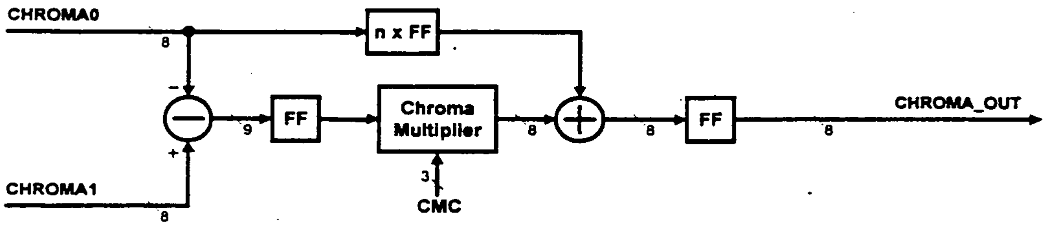


Figure C 4 -- Chroma Vertical Format ConverterOperation of the Luminance Vertical Sample Rate Converter (Y VFC)

Operation of the Luminance Vertical Format Converter

In the luminance format converter luminance pixel data is filtered in the vertical direction, using data from one to four scan lines of the source picture to generate each line of the converted picture. Data is obtained from the on-chip 128 x 960 luma RAM which is also used for block to line reordering. When processing 1920 x 1080 HD images in full resolution, the entire RAM is used in performing the reordering function; only one line-ordered signal is available, and vertical format conversion is not possible. When other formats are processed, or when processing 1920 x 1080 in an SD application using H/2 compression, sufficient RAM space is available to implement extra line memories in additon to the reordering memory, producing up to four line-ordered outputs.

The signal flow portion of the luminance vertical format converter is shown in Figure 1. The inputs LUMA0 - LUMA3 receive data from the four line memory outputs. The filter is a "poly-phase" filter; i.e. the weights or coefficient values change for each line of output data, producing the multiple "phases" of the filter. This requires changing the filter characteristics on a line-by-line basis. The configuration of the filter is determined by a 57 bit control word. A set of 16 64 bit registers allows the storage of up to 16 different control words from the host bus. An additional host bus register specifies the total number of line delays to be used in a particular mode. The contents of the control word are shown in the following table.

Vertical Format Converter Control Word Contents				
Signal Description	Signal Name(s)	bits/sig	signals	Total bits
Internal Mux Control	MC1 - MC7	2	7	14
Internal Shift Control	SC1 - SC9	3	9	27
Add/Subtract Control	ASC1 - ASC4	1	4	4
Chroma Multiplier Control	CMC	4	1	4
Luma Line Memory Enable Flags	LLENAB	4	1	4
Luma Line Protect	LLPROT	3	1	3
Chroma Line Protect	CLPROT	1	1	1
Last Phase Flag	LPF	1	1	1
Total per control word			25	58

The first three signals control the luminance vertical format converter. The fourth signal sets the interpolation value of the chroma vertical format converter. The four luma line memory enable flags specify which of the line memories are to be read during each output line period. The luma line protect signal specifies the number of a line memory which is to be protected from overwriting; i.e. the process which writes data to the line memories is prevented from writing past where that memory is being read. Since the write process writes the line memories in numeric sequence, this synchronizes the writing of data to the memories; normally it would specify a line memory whos contents are being used for the last time. A chroma line protect signal performs an equivalent function for synchronizing the writing of the two chroma line memories. A last phase flag (LPF) signals the last phase of the filter; when LPF=1 the next control word is fetched from the beginning of the list, corresponding to the first phase for that mode.

The filtering is obtained by summing weighted amounts of the four input signals. Instead of using general purpose multipliers, shifting blocks are used which can shift the data from zero to 4 binary digit positions (as specified by SC1...SC7); this corresponds to multiplication by 1, 2, 4, 8, or 16. If shifted versions of the same signal are added or subtracted, many other values of filter coefficients can be generated. There are seven of these shifter paths in the luma format converter section; by setting the appropriate internal mux control (MC1-MC7), one of the four signal sources can be specified as the signal source for each.

After the shifted data has been accumulated, there are two additional shifters (also with a range of 1x to 16x). These can be added or subtracted to produce an overall gain factor for the signal; this is used to scale the signal to approximately the correct level. The signal is then divided by 64 and limited to a standard 8-bit range of 0 to 255.

Operation of the Chrominance Vertical Format Converter

The chrominance vertical format converter is shown in figure 2. This is a relatively simple two-tap linear interpolator; i.e. it uses pixel data from two source picture lines to produce a value intermediate between the two pixel values, representative of a pixel which would be placed at a corresponding position. The intermediate values can be set with a resolution of 1/8 of a line. The CHROMA0 pixel value is subtracted from the CHROMA1 pixel; the difference is multiplied by one of 9 values (0, 1/8, 2/8,...,7/8, 1) and the result is added to the CHROMA0 pixel value. The multiplier value is labeled the Chroma Multiplier Control (CMC), and is obtained from 4 bits of the control word.

DESCRIPTION OF BLOCKS USED

FF Block

This is a standard edgé triggered data latch, with clock, data and enable inputs and data outputs. In the luminance section, ALL data latches (whether explicitly shown as FF blocks, or contained inside other blocks) are enabled by a common enable signal from the LUMA H-SRC section. Likewise, in the chrominance section, ALL data latches are controlled by a common enable signal from the CHROMA H_SRC section.

MUX Block

The MUX is a 4 input 8 bit wide multiplexer, controlled by the mux control signal MCx. For each of the seven shifter paths there is a MUX which selects from the sources LUMA0 through LUMA3.

SHIFT8 Block

This block left shifts the 8 bit unsigned binary input in[7:0] by an adjustable amount, producing the 12 bit signal out[11:0]. The 3 bit unsigned binary control signal SCx[2:0] determines the amount of shift as follows:

```
if SCx[2] == 1        shift = 4
else                  shift = SCx[1:0]
```

The shift is described by:

```
if shift ≤ k ≤ shift+7    out[k] = in[k-shift]
else                      out[k] = 0
```

SHIFT14 Block

This block acts much like the SHIFT8 block, except that the input is a 14 bit twos-compliment signed binary number, with a 16 bit signed binary output. It is assumed that the values of the input and the shift specified by SCx are such that the output will always be representable by a 16 bit twos-compliment number; i.e. no overflow checking is performed. Since the quantities involved are signed, sign extension is required. As with SHIFT8, the amount of shift = s = min(SCx, 4).

```
for k < s ;                        output[k] = 0
for s ≤ k < min( s + 13, 15 )      output[k] = input[k-s]
for s + 13 ≤ k < 15                output[k] = input[13]
                                   output[15] = input[13]
```

ADD/SUBTRACT Operator

The circle with a ± inside is an adder which can also be configured as a subtractor, depending on the state of the one bit add/subtract control signal ASCx as follows:

if ASCx = 0, then the two inputs are added together.
if ASCx = 1, then the ± input is subtracted from the + input.

This operator appears in two different contexts:

With 12 bit unsigned inputs: the output is 14 bit two's-compliment signed (one additional bit of range for positive numbers, plus an added sign bit).

With 16 bit signed inputs: the output is also 16 bit two's-compliment signed (it is assumed that the input values are such that overflow never occurs following addition or subtraction).

1/16 and 1/4 Blocks

These blocks simply divide their inputs by truncation. The 1/16 block drops the 4 least significant bits of its input. The 1/4 block drops the 2 least significant input bits.

Limit to 0<x<255 Block

This block limits a 10 bit signed input to a 8 bit unsigned result as follows: for k = 0...7

```
if input[9] = = 1          output[k] = 0
else if input[8] = = 1     output[k] = 1
else                       output[k] = input[k]
```

Chroma Multiplier Block

This block multiplies in[8:0] (a 9 bit twos-compliment signed input) by CMC[3:0] (a 4 bit unsigned binary control with a range of 0 to 8 inclusive). The multiplication results in a 12 bit product which is converted to a 9 bit signed number by rounding towards zero; the sign bit is then deleted, producing out[7:0]. The lack of sign information would normally result in an ambiguity, but in this particular application the sign bit is not used.

```
if CMC[3] = = 1      out[7:0] = in[7:0]

else                 tempA[11:0] = in[8:0] x CMC[2:0]       (9 bit x 3 bit integer multiply)
                     tempB[11:0] = tempA[11:0] + 3 + in[8]        (round toward zero)
                     out[7:0] = tempB[10:3]
```

n x FF Block

This is a cascade of n flip-flops to compensate for the fixed delay through the multiplier. The value of n should be the number of clock delays in the multiplier block plus 1.

C5.5. Raster Generator

The Raster Generator block is shown in Figure C 2 and contains a horizontal (pixel) counter with a comparator and a vertical counter with a comparator.

The horizontal counter (H_CNT) is a 12-bit counter which is reset by any of the following signals:
(1) RST - master chip (asynchronous) reset.
(2) H_DRIVE_RST - from H_DRIVE_IN, a horizontal (synchronous) reset derived external to the HD-MPEG decoder.

(3) H_RST - (synchronous) automatically generated when H_CNT reaches the CLOCKS_PER_LINE bus value.

The 12-bit counter state, PIX_CNT goes to a horizontal comparator block and to other sections of the chip.

In addition to PIX_CNT, the following are bus control register inputs to the horizontal comparator:
(1) HDO - H_DRIVE (horizontal drive) start.
(2) HDS - H_DRIVE stop.
(3) XDO - H_ACTIVE_PIX (horizontal active pixel timing signal) start.
(4) XDS - H_ACTIVE_PIX stop.
(5) CLOCKS_PER_LINE - Total number of clock intervals (active + blanking) in a line period.

The outputs of the horizontal comparator block are: H_RST, H_DRIVE, H_ACTIVE_PIX, and HALF_LINE (a pulse which occurs twice a line when PIX_CNT = CLOCKS_PER_LINE or CLOCKS_PER_LINE/2.

The vertical counter (V_CNT) is a 12-bit counter which is reset by any of the following signals:
(1) RST - master chip reset.
(2) V_DRIVE_RST - from V_DRIVE_IN, a vertical reset derived external to the chip, or set by a bus register written by the microprocessor.
(3) H_RST - automatically generated when H_CNT reaches the CLOCKS_PER_LINE bus value.

The 12-bit counter state, HALF_LINE_CNT goes to a vertical comparator block and to other sections of the chip.

In addition to HALF_LINE_CNT, and H_RST, the following are bus control register inputs to the vertical comparator:
(1) VDO - V_DRIVE (vertical drive) start.
(2) VDS - V_DRIVE stop.
(3) YDO - V_ACTIVE_PIX (vertical active pixel timing signal) start.
(4) YDS - V_ACTIVE_PIX stop.
(5) HALF_LINES_PER_VERTICAL - the number of HALF_LINE pulses per vertical period, same as the total number of lines per frame (an odd number means an interlace display, an even number means a progressive display).

The outputs of the vertical comparator block are: V_RST, V_DRIVE, V_ACTIVE_PIX, EnotO (even not odd field), and LINE_CNT (line count - of a frame).

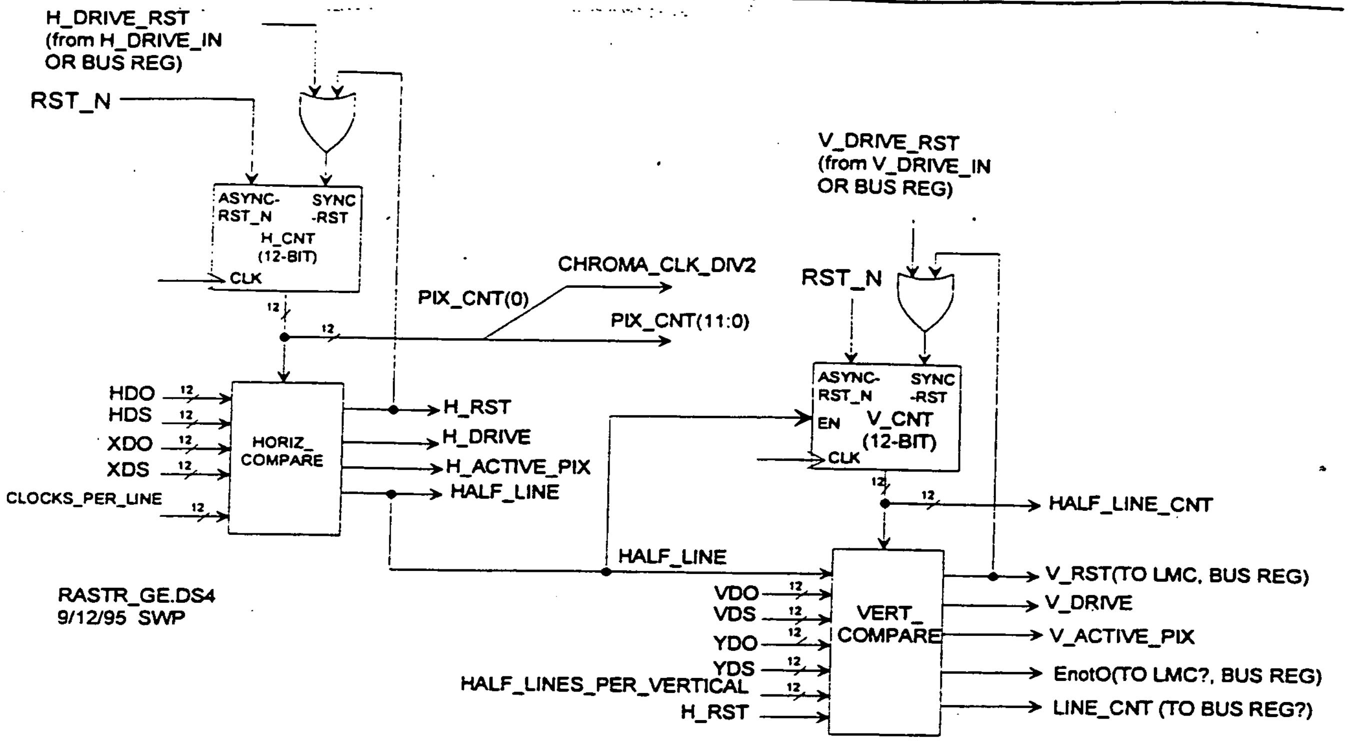


Figure C ?. Raster Generator

C5.6. LMU Line Doubler

C5.6.1. General

The LMU line doubler is shown in Figure C 5. This block calculates the extra luma lines needed to generate a 1080 line display from a 480 line input. For each field, lines corresponding to that field are passed right through the block. In other words, luma2H is the same as line263. Lines corresponding to the previous field must be estimated by the LMU algorithm based upon the amount of motion estimated from the previous field and frame. If the motion values are zero, then the line from the previous field is used. The more motion present, the more the average of the lines above and below the current line from the current field is used.

When the LMU line doubler is in use, the local memory controller (LMC) must deliver the following signals to the display section:

(1) line0 - the current video line
(2) line262 - a video line delay 262 lines (nearly a field) from line0
(3) line525 - a video line delay 525 lines (a full frame) from line0
(4) mot262 - a motion signal (stored with the same format as video) delayed 262 lines from mot0, the signal which was written to external memory.
(5) mot525 - a frame delay (525 lines) of mot0.

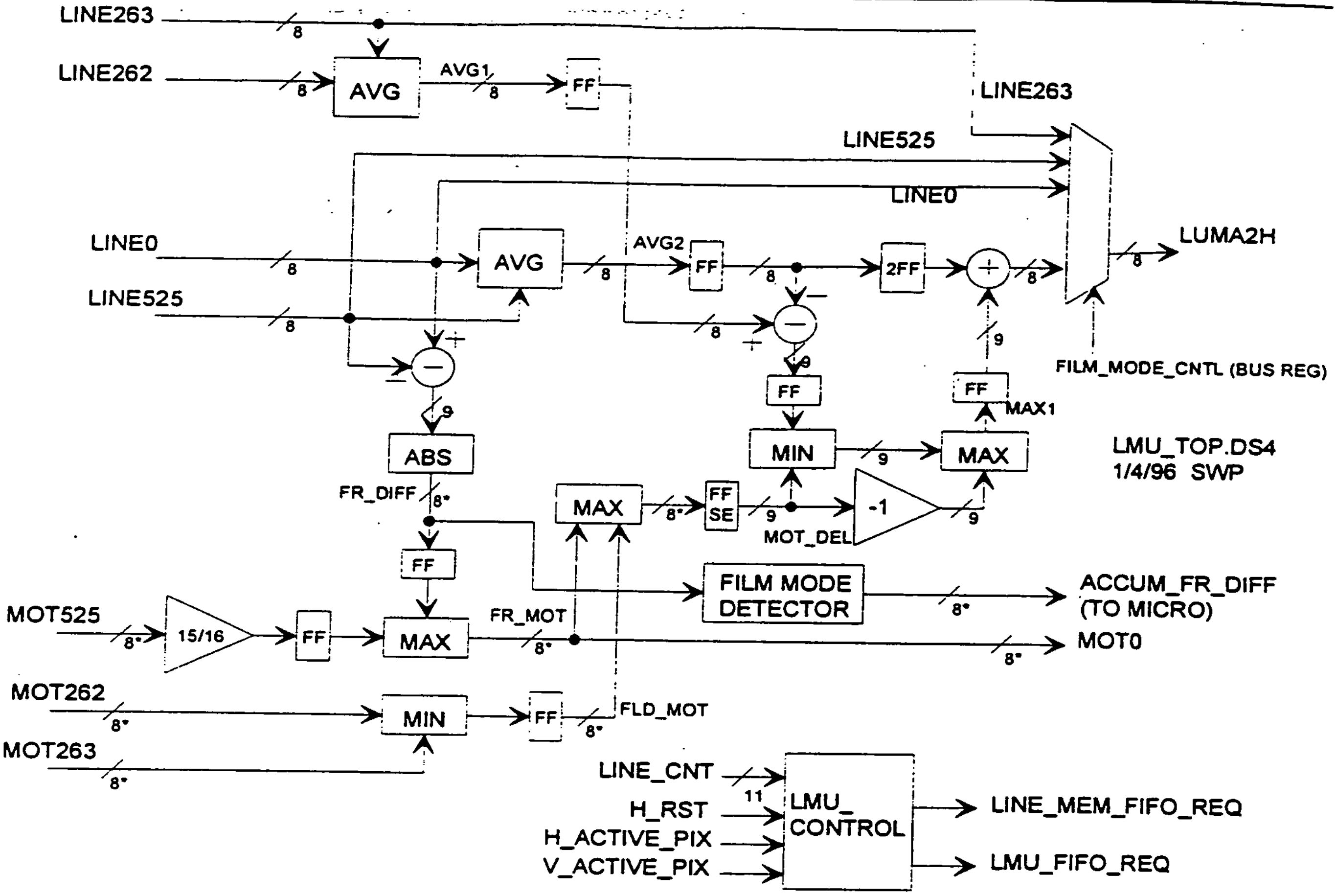


Figure C 5. LMU Line Doubler with Film Mode Detection

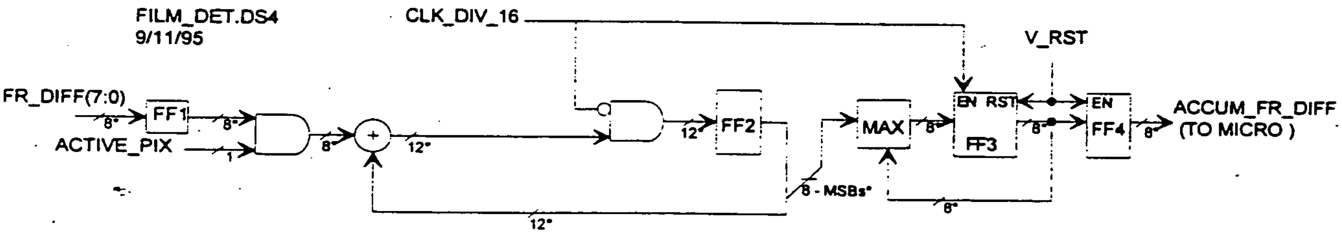


Figure C 6. Film Mode Detector

C5.6.2. Film Mode Detector

The film mode detector's basic function is to accumulate frame differences over an entire field, and pass onto the micro an 8-bit value representing the frame difference. But, a simple accumulation over the field does not result in the most effective measure of frame differences for the purposes of detecting film frame boundries. The following is the preferred algorithm:

> The frame difference signal (FR_DIFF, 9-bit, unsigned) is gated by ACTIVE_PIX = HOR_ACTIVE_PIX and VERT_ACTIVE_PIX. this insures that the accumulation is only made over the active video area. The signal is than accumulated over 16 pixels, where the accumulator output is limited to 16 bits. The signal CLK_DIV_16 is high one clock period out of 16, and holds the final output (in FF2) while then clearing the accumulation loop once every 16 clocks.
> The 8 MSBs of the accumulated signal are compared with the maximum found so far over the field. The updated maximum is held by the FF3. This field maximum is transferred to FF4 and the value in FF3 cleared by the vertical reset signal (V_RST).

C5.7. Horizontal Sample Rate Converter (H-SRC)

C5.7.1. General

The luma and chroma H-SRCs are both 4 tap FIR (polyphase) filters, with 16x oversampling, using "sparse" coefficient sets such that the number of adders per tap is about 2. The luma and chroma H-SRCs are preceeded by pre-filters which improve the overall frequency response. The pre-filters are fixed filters (5 tap) with two switchable coefficient sets. One set results in an overall response for the SRC which is approximately flat; the other set results in a response which has approximately +2 dB of peaking. The data rate through the filters is 81 MHz.

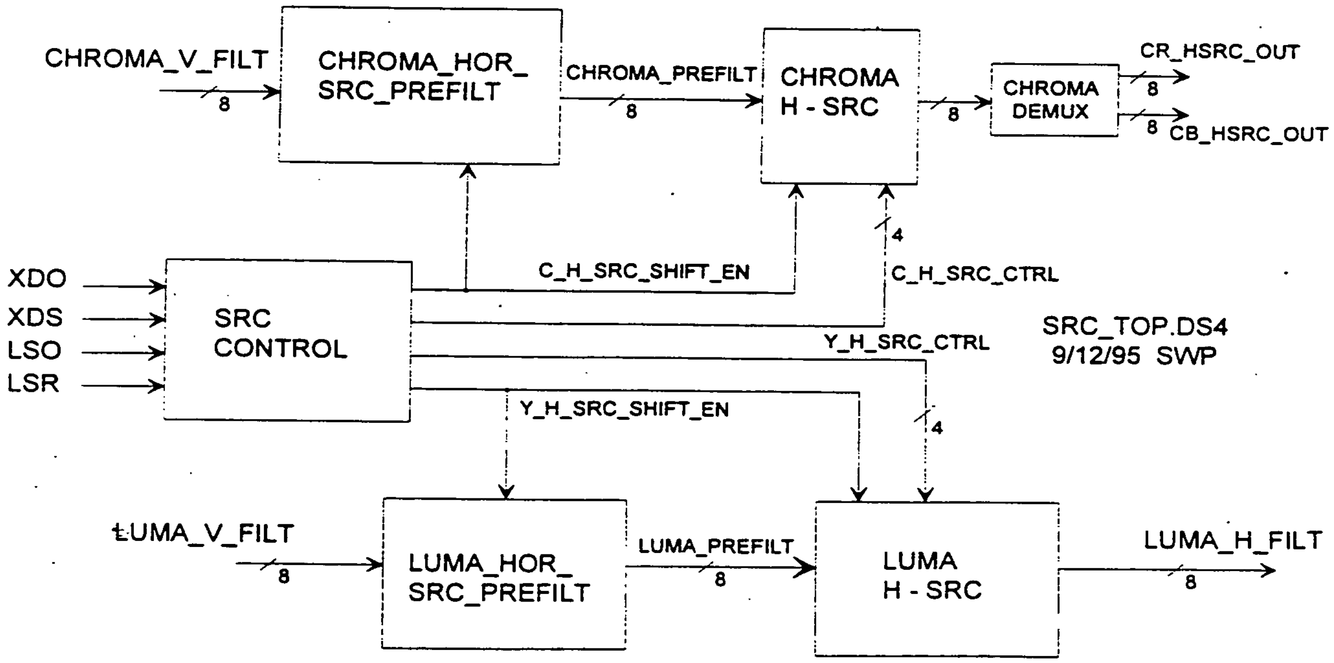


Figure C 6. Sample Rate Converter (SRC) Top Level

C5.7.2. Luma Prefilter

The luma prefilter is a 5-tap, symmetric filter with coefficients C0,C1,C2, followed by a simple [1 1] filter (as shown in Figure C 7). The coefficients are switched using the control signal Y_PEAK_MODE. When Y_PEAK_MODE = 1, the overall filter response is chosen to be peaked by +2 dB, when Y_PEAK_MODE = 0, the overall filter response is approximately flat. Note that the pre-filter contains two internal divide-by-2 truncations, and the output is 9 bits (one bit more than the input). The Z-transforms of the pre-filters are shown below:

Flat Response PreFilter: $$\frac{1+z^{-1}}{2}\cdot\frac{1-5z^{-1}+12z^{-2}-5z^{-3}+z^{-4}}{4}$$

Peaked Response PreFilter: $$\frac{1+z^{-1}}{2}\cdot\frac{2-9z^{-1}+18z^{-2}-9z^{-3}+2z^{-4}}{4}$$

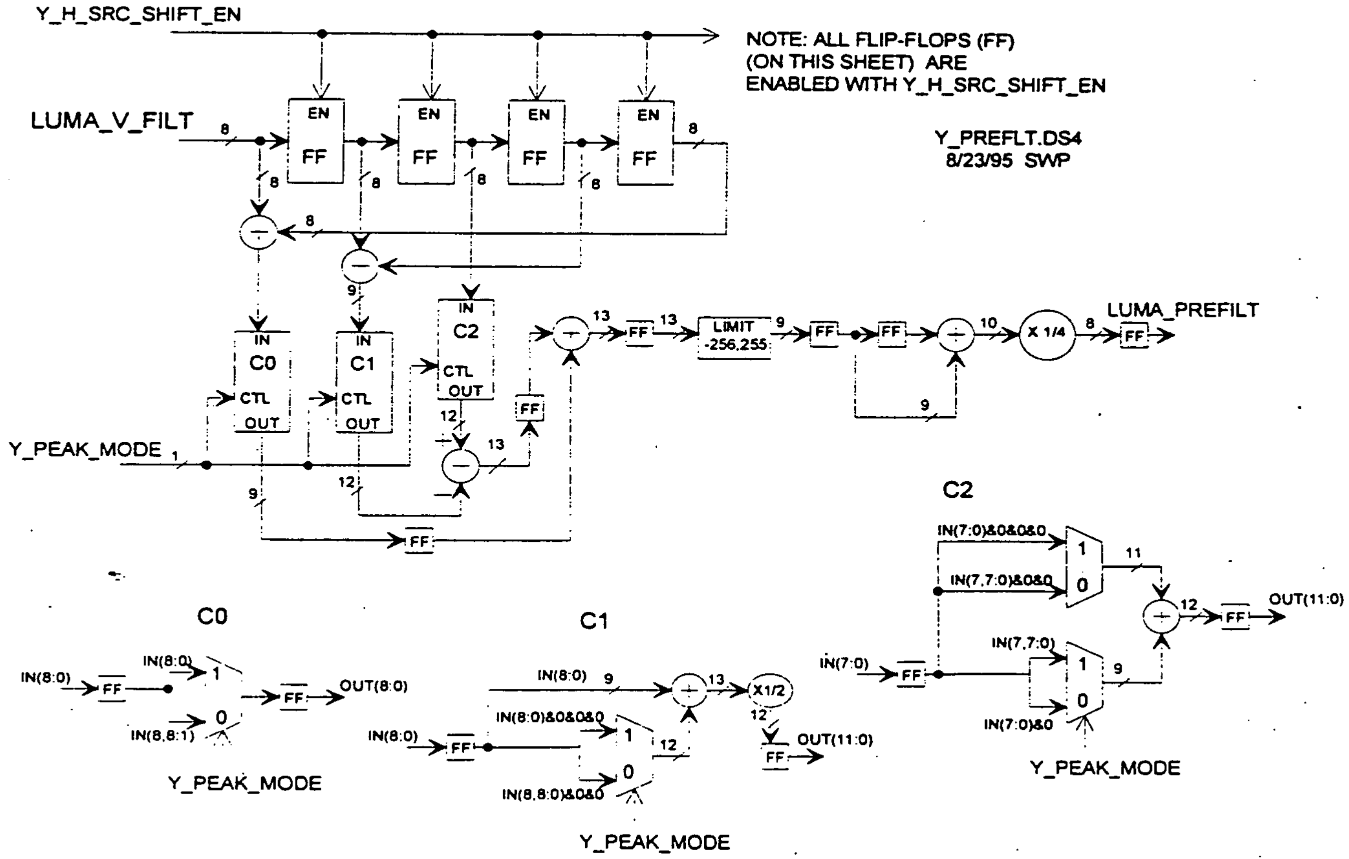
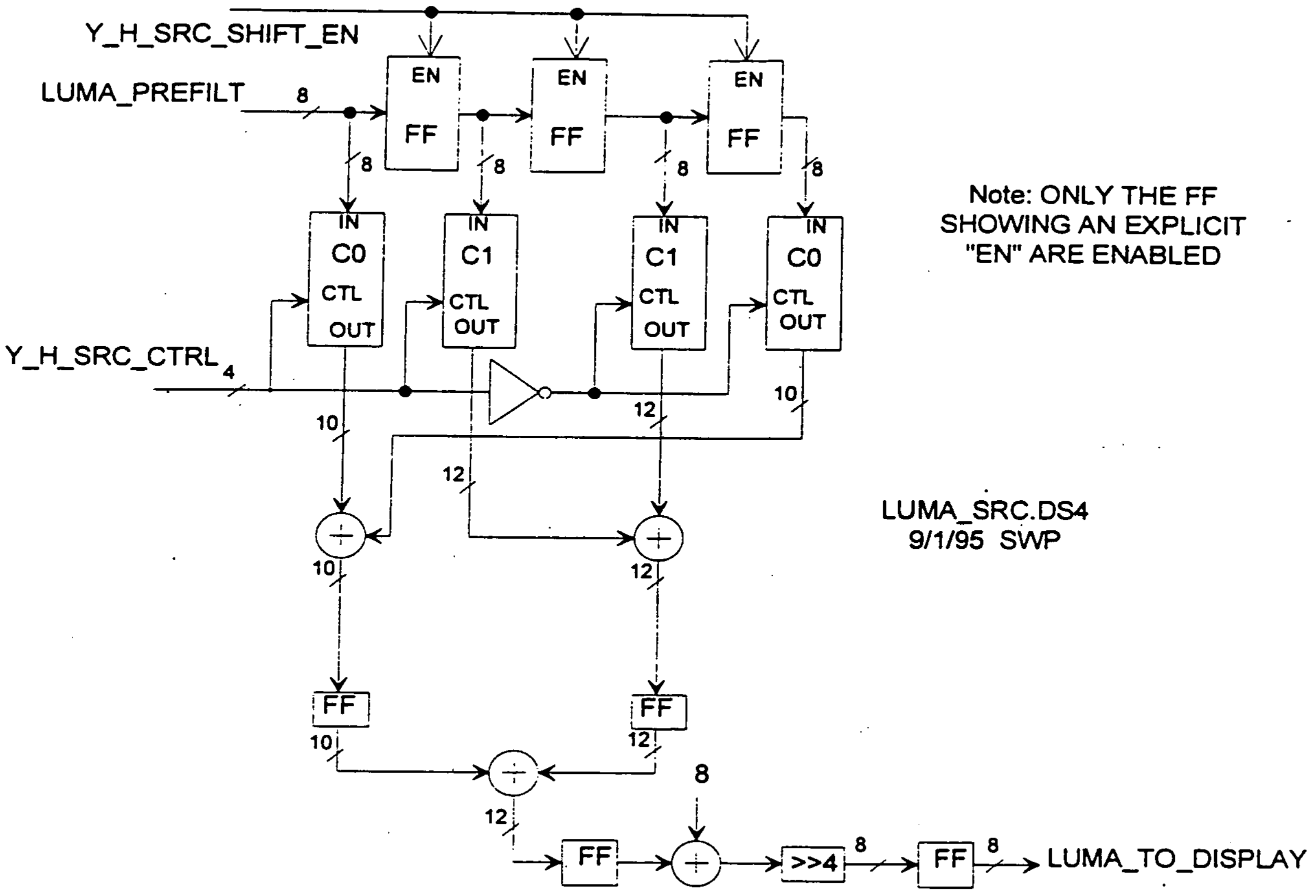


Figure C 7. Luma Horizontal SRC Prefilter



Note: ONLY THE FF
SHOWING AN EXPLICIT
"EN" ARE ENABLED

LUMA_SRC.DS4
9/1/95 SWP

Figure C 8. Luma Horizontal SRC

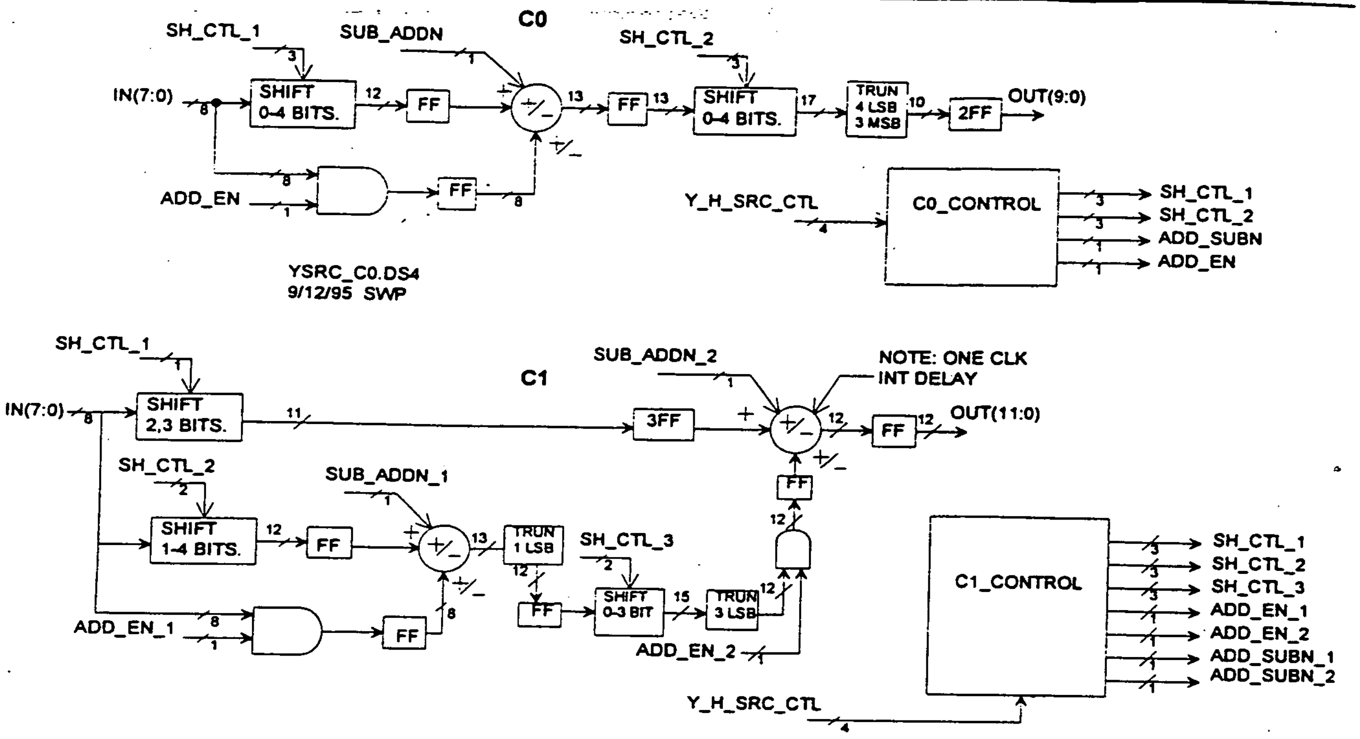


Figure C 9. Luma H - SRC Coefficients

C5.7.3. Luma Horizontal SRC

The luma H-SRC (shown in Figure C 8) takes the 8-bit output of the prefilter and performs sample rate conversion. The values of the coefficients are shown below in Table C 2. The first column in Table C 2 is the Y_H_SRC_CTL(3:0) signal which controls the phase of the filter on a pixel by pixel basis. The second column shows the actual effective delay through the filter for each of these phases. The next four columns show the tap weights of the four taps in the actual converter. The first two of these columns (1 for no delay, and z^{-1} for a one clock delay) actually show the values of the gain due to coefficients C0 and C1. The last two columns (z^{-2} - a two clock delay, and z^{-3} - a three clock delay) show the gains for the last two taps. These gains (as indexed by Y_H_SRC_CTL(3:0)) are the same as the gains due to C0 and C1 if the control signal Y_H_SRC_CTL(3:0) has the bits inverted. Hence the four tap gains can be computed by using two copies of the C0 coefficient block and two copies of the C1 coefficient block. With the second C1 and C0 having an inverted control signal (Y_H_SRC_CTL(3:0)).

COMPENSATED 4 TAP SAMPLE RATE CONVERTER
4 Tap SRC(Sum Coefficients=256):

Y/C_H_SRC_CTL	Delay	1	z^{-1}	z^{-2}	z^{-3}
15	63/32	1	62	137	56
14	61/32	2	68	138	48

13	59/32	4	74	138	40
12	57/32	5	81	136	34
11	55/32	7	88	131	30
10	53/32	8	96	128	24
9	51/32	10	104	122	20
8	49/32	15	108	116	17
7	47/32	17	116	108	15
6	45/32	20	122	104	10
5	43/32	24	128	96	8
4	41/32	30	131	88	7
3	39/32	34	136	81	5
2	37/32	40	138	74	4
1	35/32	48	138	68	2
0	33/32	56	137	62	1

TABLE C 2. Horizontal Sample Rate Converter Coefficients

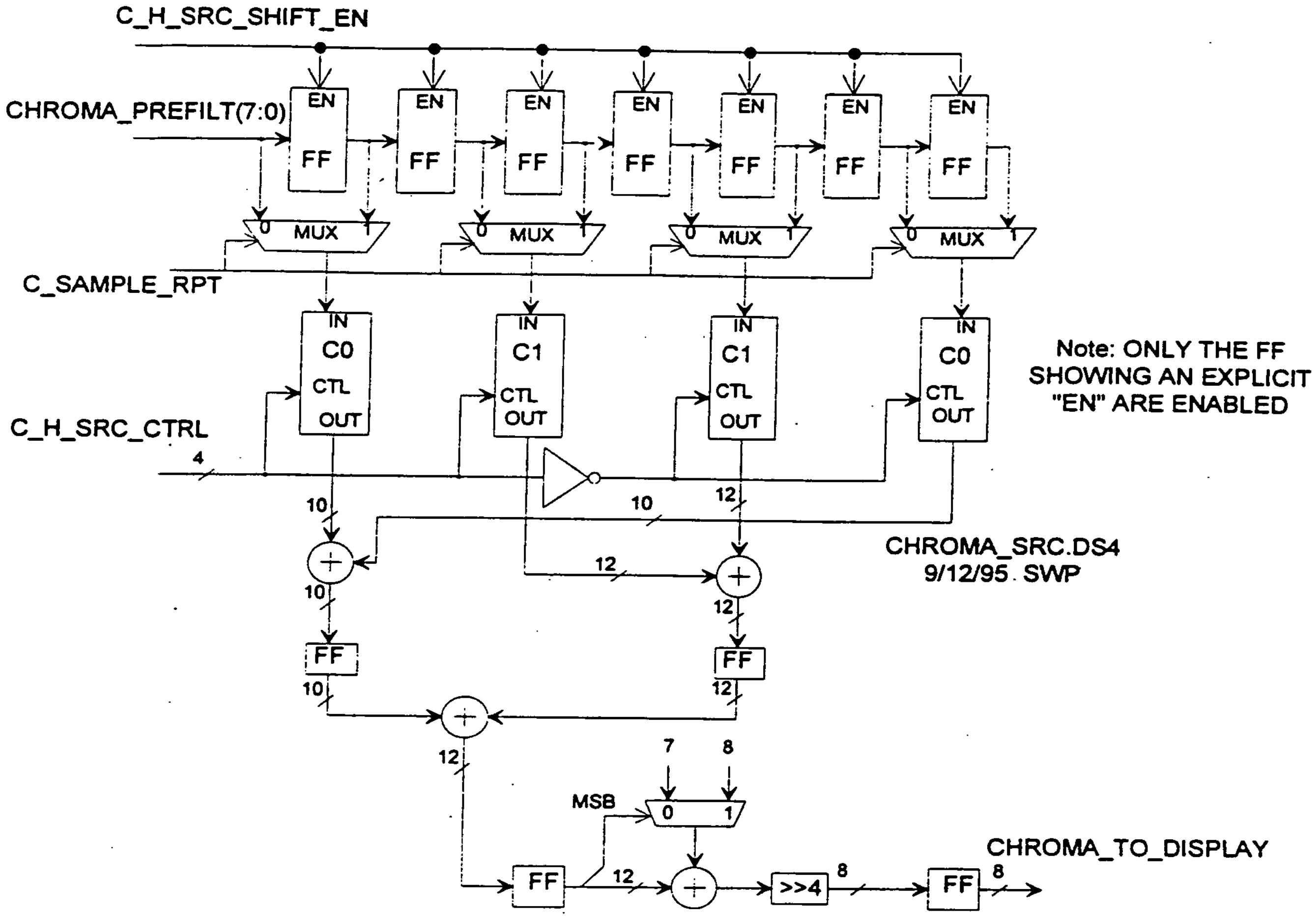
Table C 3 (below) shows the cannonic signed digit representation of the tap weights. This table shows the bit shifts and addition/subtraction necessary to implement the coefficient gains shown in Table C 2 (above). For example, consider the table entry for Delay 55/32 (Y_H_SRC_CTL = 11) : 00000+00- . This entry can be interpreted as: the input shifted left (up) 3 bits (times +1) is added to the input shifted up zero bits (times -1). The 3-bit shift up implements a gain of 8, and the subtraction of 1 yields a net gain of 7. Note that this is the gain for Y_H_SRC_CTL = 11 shown in Table C 2.

The coefficient calculations for C0 and C1 are shown in Figure C 9 (above) to implement the shifts implied by Table C 3. Coefficient C0 is implemented using two 0 to 4 bit shifts, an adder/subtracter, and an "AND" gate to enable the adder. The control signals for C0 are generated by the C0_CONTROL block from the phase signal Y_H_SRC_CTL. The VHDL code for C0_CONTROL must be written to implement the shifts described in Table C 3. Coefficient C1 is implemented using three three variable shifters, two adder/subtractors, and two "AND" gates to enable the adders. The control signals for C1 are generated by the C1_CONTROL block from the phase signal Y_H_SRC_CTL.

Cannonic Signed Digit Representation:

Y/C_H_SRC_CTL	Delay	1	z^{-1}	z^{-2}	z^{-3}
15	63/32	00000000+	00+0000-0	0+000+00+	00+00-000
14	61/32	0000000+0	00+000+00	0+000+0+0	00+0-0000
13	59/32	000000+00	00+00+0+0	0+000+0+0	000+0+000
12	57/32	000000+0+	00+0+000+	0+000+000	000+000+0
11	55/32	00000+00-	0+0-0-000	0+0000+0-	000+000-0
10	53/32	00000+000	0+0-00000	0+0000000	000+0-000
9	51/32	00000+0+0	0+0-0+000	0+000-0+0	0000+0+00
8	49/32	0000+000-	0+00-0-00	0+00-0+00	0000+000+
7	47/32	0000+000+	0+00-0+00	0+00-0-00	0000+000-
6	45/32	0000+0+00	0+000-0+0	0+0-0+000	00000+0+0
5	43/32	000+0-000	0+0000000	0+0-00000	00000+000
4	41/32	000+000-0	0+0000+0-	0+0-0-000	00000+00-
3	39/32	000+000+0	0+000+000	00+0+000+	000000+0+
2	37/32	000+0+000	0+000+0+0	00+00+0+0	000000+00
1	35/32	00+0-0000	0+000+0+0	00+000+00	0000000+0
0	33/32	00+00-000	0+000+00+	00+0000-0	00000000+

TABLE C 3. Horizontal Sample Rate Converter CSD Representation
(CSD = Cannonic Signed Digit)



Note: ONLY THE FF SHOWING AN EXPLICIT "EN" ARE ENABLED

CHROMA_SRC.DS4
9/12/95 SWP

Figure C 11. Chroma Horizontal SRC

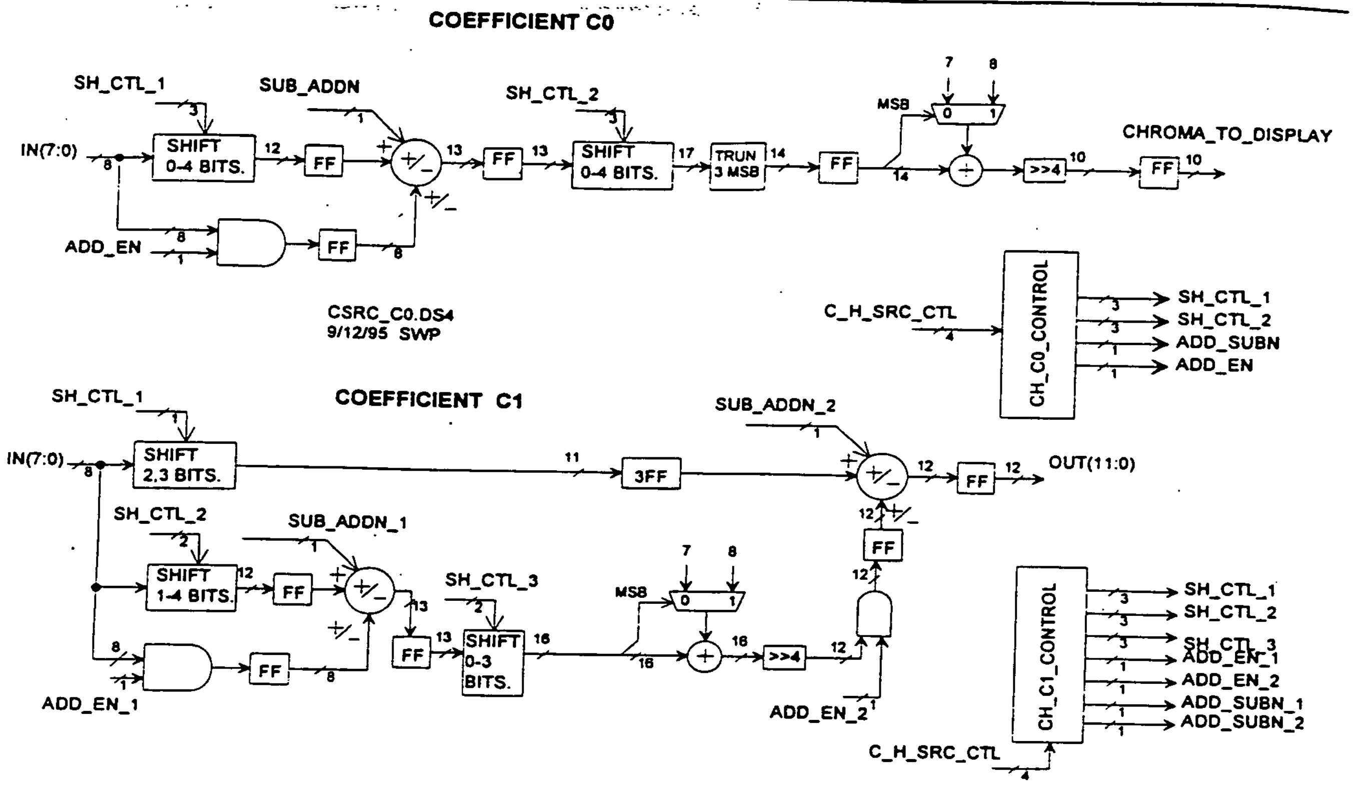


Figure C 12. Chroma H - SRC Coefficients

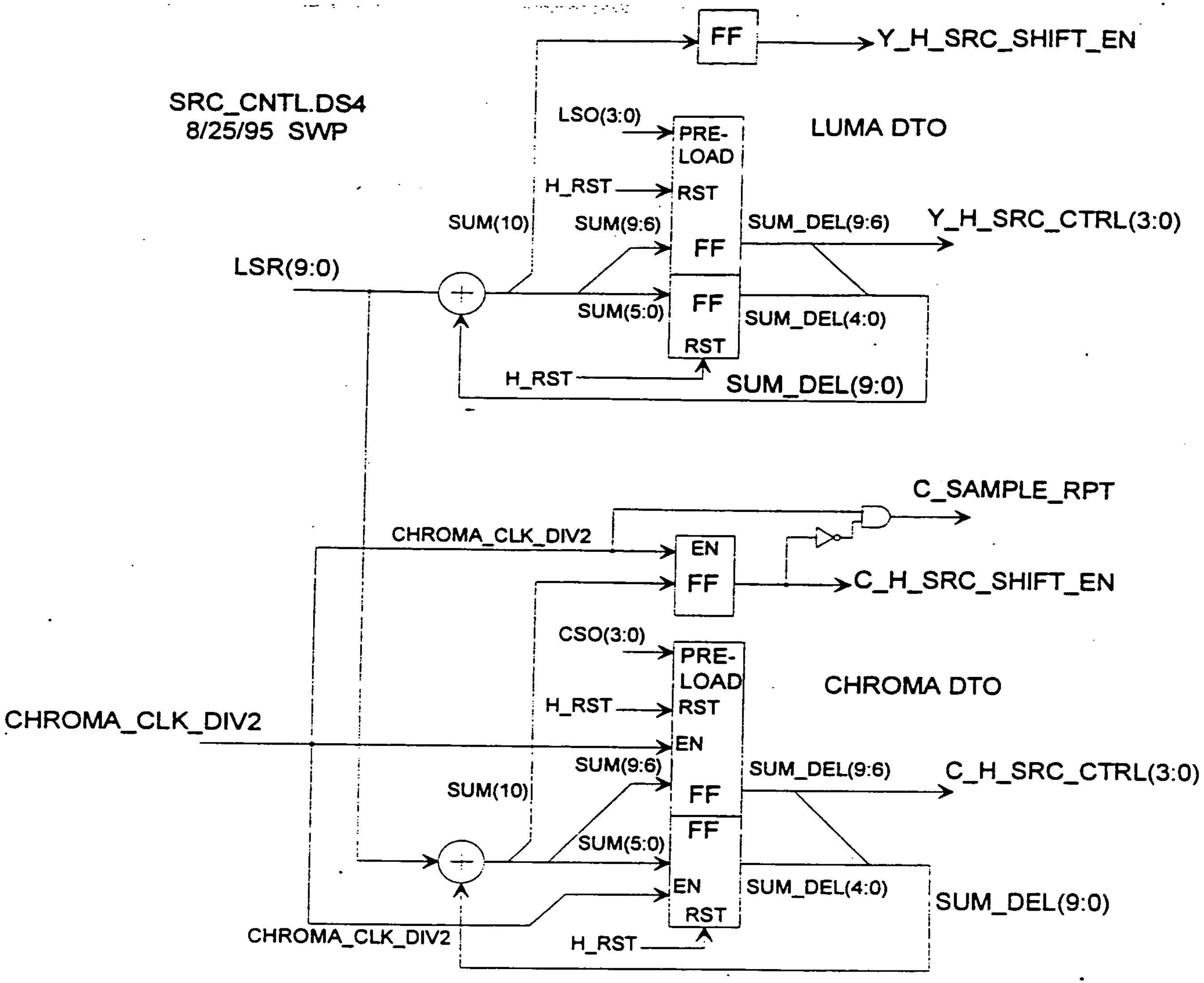


Figure C13. SRC CONTROL

C6. APPLICATIONS BUS REGISTERS

C6.1. Display Registers (Unformatted)

REGISTER NAME	DESCRIPTION/COMMENTS	# bits	LOCATION	Double Buffer
HDO	Start of hor. drive relative to H_RST	11	Raster Gen	Vsync
HDS	Stop of hor. drive relative to H_RST	11	Raster Gen	Vsync
XDO	Start of active video relative to H_RST	11	Raster Gen	Vsync
XDS	Stop of active video relative to H_RST	11	Raster Gen	Vsync

clocks_per_line	Period of H_RST, in display clock domain	11	Raster Gen	Vsync
VDO	Start of vert. drive relative to V_RST	11	Raster Gen	Vsync
VDS	Stop of vert. drive relative to V_RST	11	Raster Gen	Vsync
YDO	Start of active video relative to V_RST	11	Raster Gen	Vsync
YDS	Stop of active video relative to V_RST	11	Raster Gen	Vsync
PAN	(horizontal) Pan vector - integer part	11	Y/C RAM control	Vsync
SCAN	(vertical) Scan vector higher order bit controlled in LMC	3	Y/C RAM control	Vsync
H_SRC_BYP	bypass the horiz. sample rate converter	1	H - SRC	
Y_VFC_BYP	bypass the vertical format converter	1	Y_VFC	
LMU_active	Enables the LMU mode	1	LMU/OSD	
LSO_DISP	Horizontal initialization for luma H-SRC (for sub-pix resolution of PAN).	4	Y - H - SRC	
CSO_DISP	Horizontal initialization for chroma H-SRC (for sub-pix resolution of PAN).	4	C - H - SRC	
half_lines_per _vertical	Also, total number of lines per frame: odd => interlaced, even => prog. scan	11	Raster Gen	Vsync
BLOCKPAIRS_PER_ LINE	Number of pairs of 8x8 blocks per line (decompressed), decomp. clock domain	7	BTL Add Gen	
BTL_MODULUS	8*Blockpairs_per_line -1	10	BTL Add Gen	
CBTL_MODULUS	4*Blockpairs_per_line -1	10	CBTL Add Gen	
OUTPUT_READ_ LINE0_START	Starting address of line memory storage (luma RAM)	10	BTL Add Gen	
COUTPUT_READ_ LINE0_START	Starting address of line memory storage (chroma RAM), 3 LSB's trunc?	8	CBTL Add Gen	
LSR	upsampling factor 1024/LSR	10	SRC-Ctrl	
LMU_LINE0_ START(9:2)	Starting address of LMU line memory storage	8	BTL Add Gen	
NUM_OUTPUT_ LINE_MEM	Number of output line memories used (luma) 0 to 4 allowed	3	BTL Add Gen	
DISPLAY_REFRESH _INTERVAL	Number of RAM_CLK cycles between display RAM (Y & C) refreshes	8	BTL Add Gen	
DISPLAY_REFRESH _INCREMENT	RAM Address increment for refresh	8	BTL Add Gen	
RASTER_GEN_RST	Reset RASTER_GEN pixel and line counters	1	Raster Gen	
ACCUM_FR_DIFF	Accumulated frame difference to detect film (3:2 pulldown) phase (READ reg.) Should be read every field by micro when D1 interface is active	8	LMU	
FILM_MODE_CNTL	Forces field jamming and controls which field is jammed (to implement film mode).	2	LMU	
VFC_INSTR_TOP	Circular buffer 16 words, 58bits/word; coeff. sequences for the VFC (luma & chroma): implemented 64bits/word? For top field.	8	VFC	circular
VFC_INSTR_BOT	Circular buffer 16 words, 58bits/word; coeff. sequences for the VFC (luma & chroma): implemented 64bits/word? For bottom field.	8	VFC	circular

C6.2. Display Registers (Formatted for the Applications Bus)

Application Register	Construction
HDO[10:8]	HDO[10:8]
HDO[7:0]	HDO[7:0]
HDS[10:8]	HDS[10:8]
HDS[7:0]	HDS[7:0]
XDO[10:8]	XDO[10:8]
XDO[7:0]	XDO[7:0]
XDS[10:8]	XDS[10:8]
XDS[7:0]	XDS[7:0]
VDO[10:8]	VDO[10:8]
VDO[7:0]	VDO[7:0]
VDS[10:8]	VDS[10:8]
VDS[7:0]	VDS[7:0]
YDO[10:8]	YDO[10:8]
YDO[7:0]	YDO[7:0]
YDS[10:8]	YDS[10:8]
YDS[7:0]	YDS[7:0]
PAN[10:8]	PAN[10:8]
PAN[7:0]	PAN[7:0]
SCANR[14:8]	SCAN[10:3] is located in the LMC
SCANR[7:5]	SCAN[2:0]
CLKLN[10:8]	clocks_per_line[10:8]
CLKLN[7:0]	clocks_per_line[7:0]
LSOCSO[7:0]	LSO_DISP[3:0], CSO_DISP[3:0]
HLFLN[10:8]	half_lines_per_vertical[10:8]
HLFLN[7:0]	half_lines_per_vertical[7:0]
BPPLN[6:0]	BLOCKPAIRS_PER_LINE[6:0]
CRLN0[7:0]	COUTPUT_READ_LINE0_START[7:0]
ORLN0[9:8]	OUTPUT_READ_LINE0_START[9:8]
ORLN0[7:0]	OUTPUT_READ_LINE0_START[7:0]
BTLMOD[9:8]	BTL_MODULUS[9:8]
BTLMOD[7:0]	BTL_MODULUS[7:0]
CBTLMOD[9:8]	CBTL_MODULUS[9:8]
CBTLMOD[7:0]	CBTL_MODULUS[7:0]
LSR[9:8]	LSR[9:8]
LSR[7:0]	LSR[7:0]
LMULN0[7:0]	LMU_LINE0_START[9:2]
DRINT[7:0]	DISPLAY_REFRESH_INTERVAL[7:0]
VINSTT[7:0]	VFC_INSTR_TOP[7:0]
VINSTB[7:0]	VFC_INSTR_BOT[7:0]
DRINCR[7:0]	DISPLAY_REFRESH_INCREMENT[7:0]
ACCFRDIF[7:0]	ACCUM_FR_DIF[7:0]
DCTRL[5]	H_SRC_BYP[1]
DCTRL[4]	Y_VFC_BYP[1]
DCTRL[3]	LMU_active[1]
DCTRL[2:0]	NUM_OUTPUT_LINE_MEM[2:0]

FMCTRL[1:0]	FILM_MODE_CNTL[1:0]
DRST[0]	RASTER_GEN_RST[0]

C7. TEST STRUCTURES

CMD[3:0]

Command Register

Address: 61
Data Type: Byte
Reset: 0

Register Type: R/W
Double Buffer: Action W

Composition:
CMD[3] : select intra (not inter) quantization table for loading
CMD[2] : launch pipeline reset
CMD[1] : launch decoding software reset
CMD[0] : restart header search

Description:
Command register (launch actions) and QM selection

PLL_DIV_n[31:0]

PLL Divider Setup

Address: 50-53, 54-57, 58-5B (n=0,1,2)
Data Type: Byte
Reset: 0

Register Type: R/W
Double Buffer: special

Composition:
PLL_DIV[31,30,23,15,7] : don't care (or reserved bits)
PLL_DIV[29] : not reset divider bit
PLL_DIV[28] : divide by 2 flag (active high)
PLL_DIV[27:24] : P0[3:0]
PLL_DIV[22:16,14:12] : PR[9:0]
PLL_DIV[11:8,6:0] : Q[10:0]

Description:
Configures PLL fractional (P/Q) divider.

HD MPEG VIDEO DECODER

APPENDIX F

SYSTEM TIMEBASE

Thomson Consumer Electronics, Inc.
Indianapolis, Indiana, USA

Revision No. 2.2

THESE DRAWINGS AND SPECIFICATIONS ARE THE PROPERTY OF THOMSON CONSUMER ELECTRONICS, INC. AND SHALL NOT BE REPRODUCED OR COPIED OR USED AS THE BASIS FOR THE MANUFACTURE OR SALE OF APPARATUS OR DEVICES WITHOUT WRITTEN PERMISSION.

TCE PROPRIETARY AND CONFIDENTIAL

F1. OVERVIEW

The HD-MPEG IC operates with 4 primary clocks which need not be locked or originate from the same time base. The MPEG decoding module operates with a nominal 54 MHz clock and includes the VLD, Decode Pipe and other circuitry. The display section uses a clock which is locked to the MPEG system clock reference and may operate as low as 27Mhz and as high at 81Mhz, with sufficient design margin to insure a practical design. The LMC/Memory interface uses a third clock, which may operate as high as 100Mhz. Some compression modes will allow less than 100Mhz external bus speeds, thus multiplier factors must permit this and other lower memory clock speeds. The memory decompression process utilizes a fourth clock

It is anticipated that these clocks will, in most applications, originate from the same time base, however provision is made to allow completely asynchronous operation with clocking from an independent external source for each of the four primary clocks.

Two remaining secondary clocks (D1 Video input clock and the Applications bus strobes and chip selects) are not created or switched within this subsystem.

F2. CLOCK GENERATION

F2.1. Clock Functionality

The clock generator circuit provides independent source selection for each of the four primary clocks used within the HD video decoder IC. The selection source can be internal or external. An externally sourced clock can be sourced from a pin on the HD video decoder IC. An internally sourced clock is sourced from a fractional "P/Q" divider circuit fed from a classic PLL frequency multiplier. The PLL's VCO is constructed to provide a three phase clock source to the fractional dividers. The three-phase signal is used in a special de-jitter circuit that minimizes jitter due to P/Q divides to 1/3rd of the period of the VCO frequency, or less.

There are two programmable PLL's used in the design of the clock generator circuit. Each PLL circuit feeds three P/Q fractional dividers. Each fractional divide output can be the source for any of the four clocks used in the HD Video Decoder IC.

Size of divisors and multipliers shall be large enough to permit the granularity of selected frequencies to less than 100Khz. Two exceptions are 27Mhz and 27.027Mhz, both of which must be exact with regard to the source reference of 13.5Mhz and 27Mhz. A special consideration of 27Mhz and 27.027Mhz clock generation is that the application must switch between these frequencies during normal viewing operation, based on coded picture frame rates. The switch from one mult ratio to another mult ratio must be timed in such a way that minimal transients will exist: The clock circuit shall not permit a high or low pulse or pulses having a duration shorter than half the period of the selected timebase.

F2.1.1. Typical Clock Rates

Typical operation for 16x9 HD chassis assumes:
a.) Display clock ~81Mhz,
b.) VLD, Decode Pipe, and Compress clock ~54Mhz
c.) Memory clock ~100Mhz.
d.) Decompress clock ~81Mhz

Typical operation for 4x3 SD chassis assumes:
a.) Display clock ~30Mhz
b.) VLD, Decode Pipe, and Compress clock ~54Mhz
c.) LMC and Memory clock ~ 100Mhz

d.) Decompress clock ~81Mhz

Typical operation for 16x9 SD chassis assumes:
a.) Display clock ~39Mhz
b.) VLD, Decode Pipe, and Compress clock ~54Mhz
c.) LMC and Memory clock ~ 100Mhz
d.) Decompress clock ~81Mhz

Typical operation for set top box application assumes:
a.) Display and Pixel clock equals 27Mhz and 27.027Mhz, under host processor control through the applications bus.
b.) VLD, Decode Pipe, and Compress clock ~54Mhz
c.) LMC and Memory clock ~ 100Mhz
d.) Decompress clock ~81Mhz

F2.2. Internal Timebase Performance and Characteristics

The internal timebases originate from P/Q multipliers. Each multiplier incorporates a PLL composed of a phase detector, charge pump, filter, VCO, and P/Q divider circuit.

F2.2.1. VCO1 and VCO2

The reference frequency for these internal voltage controlled oscillators are derived from external reference S_Clkln. This external reference is typically 13.5Mhz or 27Mhz and is usually derived from the recovered MPEG system clock developed by the MPEG Transport Decode subsystem. Each VCO has its own isolated power and ground, minimizing induced jitter.

F2.2.1.1. Fvco
The minimum usable value is 175Mhz. The maximum usable value is 405Mhz.

F2.2.1.2. VCO Jitter
The maximum inherent VCO jitter, Jvco, is 0.5ns.

F2.2.2. P/Q Divider

F2.2.2.1. Composition of P/Q

P/Q fractional division is composed of P0 + 1 + Pr/Q. Since there are six P/Q fractional divider systems, these are denoted as P/Q[n], for n=1 through 6. The following are relevant equations and restrictions:

P/Q[n]=P0[n] + 1 + Pr[n]/Q[n]

2 <= P0[n] <= 15

0 <= Pr[n] <= 1023

0 <= Q[n] <= 2047

Pr[n] <= Q[n]

3 <= P/Q[n] <= 17

Pr[n]/Q[n] is computed as 0 if Q[n]=0

F2.2.2.1. Jitter Due to P/Q
The jitter due to P/Q multiplication shall be zero when 3 * P/Q is an integer. When 3 * P/Q is not an integer, the value of jitter due to P/Q, Jpq shall be no more than the 1/(3 * Fvco). Total multiplier jitter of the P/Q multiplier can therefore be expressed as

Jmult = Jvco + Jpq, therefore
Jmult = Jvco, where 3 * P/Q is an integer, and
Jmult = Jvco + 1/(3 * Fvco), where 3 * P/Q is not an integer

2.2.3. CLK0, CLK1, CLK2, and CLK3

When CLK0, CLK1, CLK2, and CLK3 are derived from the internal P/Q multipliers, the maximum jitter shall be Jmult.

F2.2.3.1. CLK0 – LMC and Memory I/F Clock
Controllable frequency range shall be from 60Mhz to 120Mhz

F2.2.3.2. CLK1 – VLD, Decode Pipe, and Compress Clock
Controllable frequency range shall be from 45Mhz to 90Mhz

F2.2.3.3. CLK2 – Display Process and Pixel Clock
Controllable range shall be from 27Mhz to 90Mhz.

F2.2.3.4. CLK3 – Decompress Clock
Controllable range shall be from 45Mhz to 120Mhz.

F3. CONTROL

F3.1. CLK_SEL[0..3][7:0]

The following control functions are registered in four applications bus register CLK_SEL[0][7:0], CLK_SEL[1][7:0], CLK_SEL[2][7:0], and CLK_SEL[3][7:0], which is constructed as R/W.

CLK_SEL.0 selects functions for internal clock Clk0 and IC pin S_Clk0.
CLK_SEL.1 selects functions for internal clock Clk1 and IC pin S_Clk1.
CLK_SEL.2 selects functions for internal clock Clk2 and IC pin S_Clk2.
CLK_SEL.3 selects functions for internal clock Clk3 and IC pin S_Clk3.

Each of these selects are independent and all combinations are possible. Because of the symmetrical nature of this control, the generic case of CLK_SEL[n] shall be described for all four clock control registers.

F3.1.1. CLK_SEL[n][3:0]

The lower nibble of CLK_SEL[n][7:0], CLK_SEL[n][3:0], provides source selection for internal clocks CLK0, CLK1, CLK2, and CLK3.

CLK_SEL[n][3:0] selects the following:

X000 – No clock is selected. Power down mode is selected for all devices connected to CLK[n], where n=0:.3
X001 – Output from P/Q divider0 is selected for CLK[n].
X010 – Output from P/Q divider1 is selected for CLK[n].

X011 – Output from P/Q divider2 is selected for CLK[n].
X100 – Output from P/Q divider3 is selected for CLK[n].
X101 – Output from P/Q divider4 is selected for CLK[n].
X110 – Output from P/Q divider5 is selected for CLK[n].
X111 – Pin S_Clk[n] is selected as source

F3.1.2. CLK_SEL[n][7:4]

The upper nibble of CLK_SEL[n][7:0], CLK_SEL[n][7:4], provides for clock source selections for each of four output Pins S_Clk[n].

CLK_SEL[n][7:4] selects the following:

X000 – Pin S_Clk[n] functions as an output, with a fixed value of logic "0".
X001 – P/Q divider0 is selected for output on pin S_Clk[n].
X010 – P/Q divider1 is selected for output on pin S_Clk[n].
X011 – P/Q divider2 is selected for output on pin S_Clk[n].
X100 – P/Q divider3 is selected for output on pin S_Clk[n].
X101 – P/Q divider4 is selected for output on pin S_Clk[n].
X110 – P/Q divider5 is selected for output on pin S_Clk[n].
X111 – Pin S_Clk[n] functions as an input.

F3.2. VCO Control

F3.2.1. CKG_PLL[0..1][7:0]

CKG_PLL.0 controls PLL0 and CKG_PLL.1 controls PLL1.

CKG_PLL[a][7:0] is constructed from CKG_PLL[a][7:6], x, CKG_PLL[a][4:0].
CKG_PLL[a][7:6] forms the register CKG_PLL[a].N, and may take on any value from 0 to 3, inclusive.
CKG_PLL[a][4:0] forms the register of CKG_PLL[a].M, and may take on any value from 0 through 31, inclusive

$f_{vco[z]} = f_{in}$ (CKG_PLL[z].M + 6)/(CKG_PLL[z][N] + 1)

F3.2.2. CKG_SEL.1[3:0]

CKG_SEL is the reference source selector for PLL.1. (The reference source for PLL.0 is fixed to S_CLKIn.)

CKG_SEL[2:0] selects the following sources for PLL.1:

000 selects S_CLKIn as source
001 selects S_CLK0 as source
010 selects S_CLK1 as source
011 selects S_CLK2 as source
1XX selects S_CLK3 as source

F3.3. P/Q Control

P/Q control is accomplished through registers Po[n][3:0], Pr[n][9:0], and Q[n][10:0].

For P/Q[n] fractional dividers where n = 1, 2, and 3: $f_{P/Q[n]} = f_{vco0}$ / (P/Q[n])

For P/Q[n] fractional dividers where n = 4, 5, and 6: $f_{P/Q[n]} = f_{vco1} / (P/Q[n])$

Q[n][10:0] , P0[n][3:0], and Pr[n][9:0] are consolidated into a 16-bit register as follows:

PLL_DIV[31:0] are the 32 registers bits, bytes from lower address to higher address.

PLL_DIV[31,30,23,15,7] :	don't care (or reserved bits)
PLL_DIV[29] :	not reset divider bit
PLL_DIV[28] :	div2 flag (if set, divide by another factor of 2 the output clock)
PLL_DIV[27:24] :	P0[3:0]
PLL_DIV[22:16,14:12] :	Pr[9:0]
PLL_DIV[11:8,6:0] :	Q[10:0]

F3.4. Power Down Modes

F3.4.1. Low Power Mode

Low power mode is entered after a hard reset, or by an Applications bus write of the Clock Select bits of register CLK_SEL to a value of zero (no clock is selected) through the Applications bus. In this mode, Applications bus resisters do not lose their contents by going into or out of low power mode (there was no hard reset) and can be read or written normally.

Low Power mode is exited only through an Applications bus write of the CLK_SEL register.

F3.4.2. Reduced Power Mode

Reduced power mode is entered only by a host controller reset of Clk3_Sel to zero, Clk2_Sel to zero, and Clk1_Sel to something other than zero through the Applications bus. In this mode, Applications bus registers do not lose their contents by going into or out of low power mode (there was no hard reset) and can be read or written normally. Additionally, SDRAM can be read or written, normally, through Applications bus read or writes.

Reduced Power Mode is exited only through an Applications bus write of the CLKSEL register.

F3.4.3. Display Power Mode

Display power mode is entered only by an Applications bus write of the CLKSEL register. In this mode Clk1 and Clk3 are enabled. When in Display Power Mode, display of OSD's and active D1 Video is possible.

Display Power Mode is exited only through an Applications bus write of the CLKSEL register.

F3.5. Clock Conditioning Circuits

Clock signals distributed to CLK0, CLK1, CLK2, and CLK3 are "conditioned" to prevent glitches or invalid signals that might otherwise cause a circuit to behave in an indeterminate manner.

F4. BLOCK DIAGRAM

The following figure shows the functional equivalent of the clock generator circuit. Multipliers have dedicated supply pins isolated from supplies in the remainder of the IC.

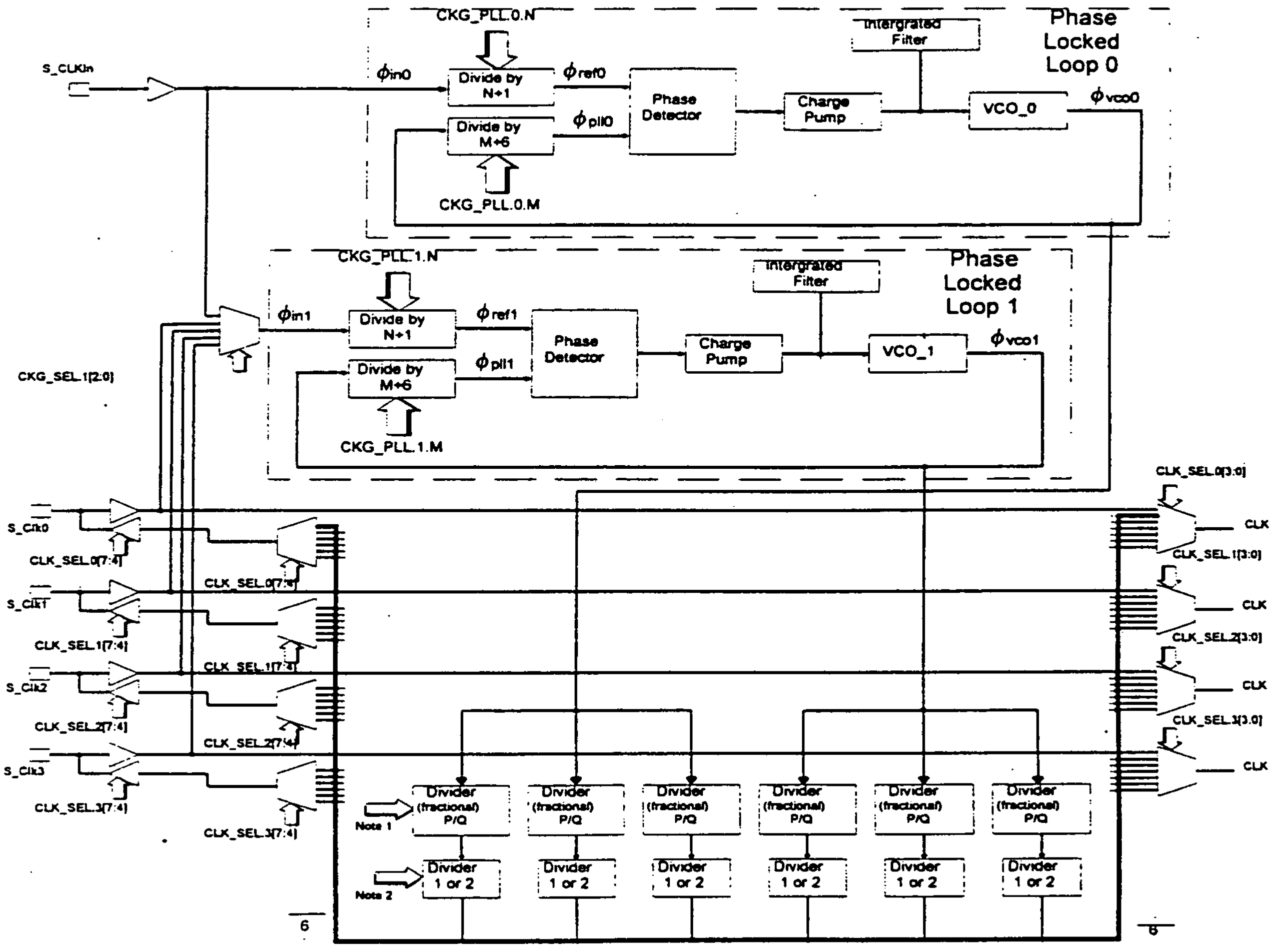


Figure F1

HD MPEG VIDEO DECODER

APPENDIX D

ON-SCREEN DISPLAY

Thomson Consumer Electronics, Inc.
Indianapolis, Indiana, USA

Revision No. 4.0

THESE DRAWINGS AND SPECIFICATIONS ARE THE PROPERTY OF THOMSON CONSUMER ELECTRONICS, INC. AND SHALL NOT BE REPRODUCED OR COPIED OR USED AS THE BASIS FOR THE MANUFACTURE OR SALE OF APPARATUS OR DEVICES WITHOUT WRITTEN PERMISSION.

TCE PROPRIETARY AND CONFIDENTIAL

D1. OVERVIEW

The table below outlines the OSD specifications and functions.

FUNCTION	DEFINITION
OSD Block	each OSD block defines a rectangular portion of the screen
OSD Block Limits	maximum 1 OSD block per horizontal line.
Display Frame Format	current raster (separate bottom and top field OSD bit maps)
Position	row = 0-1080; column = 0-1920 (OSD blocks on top pixel boundaries)
Bit Map Pixel Resolution	2 bits/pixel (four colors) OR 4/bits/pixel (16 colors)
Color Resolution	4 palettes with 14 bit YCbCr values each(for 2bits/pixel) 16 palettes with 14 bit YCbCr values each(for 4bits/pixel) (each block has an associated 4 bit mixing factor for video/OSD blending) "True Color Format" for 4:2:2 data
Output Format	24 bit Format: Full Resolution,Half Resolution, Third Resolution 8 bit Format: Full Resolution,Half Resolution

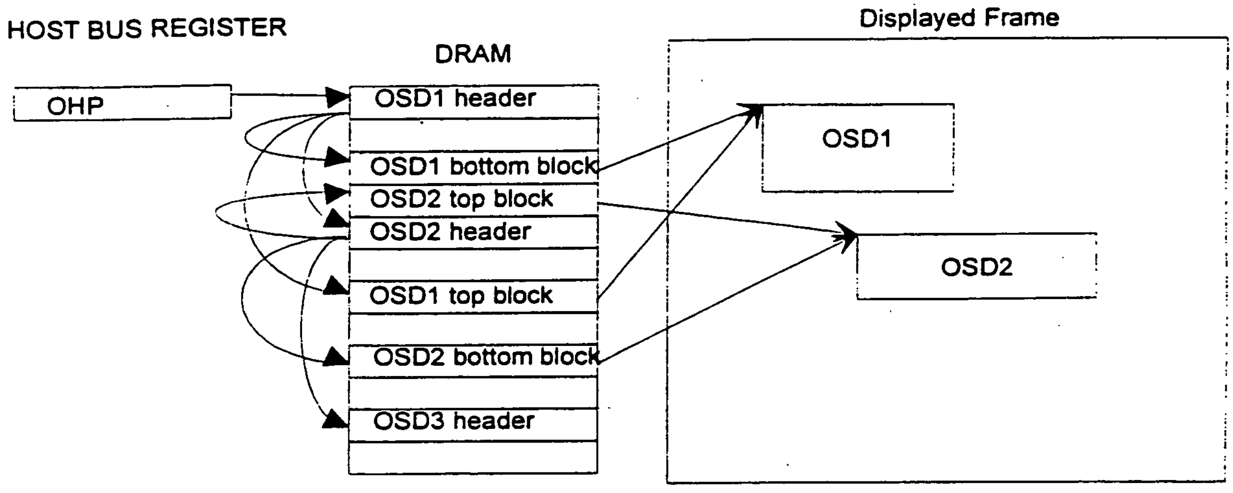
D2. GENERAL DESCRIPTION

The OSD is a separate module of the MPEG decoder that shares the same external memory. Similar to the other MPEG decoder modules, the OSD accesses the memory through the MPEG memory controller with Access Requests and an input FIFO to optimize the memory accesses.

A part of the external memory is dedicated to the OSD function. The OSD data consists of OSD header and bit-map blocks. Each header contains the start and stop positions, pointers to the start of the bottom and top field pixel bit maps, and a pointer to the next header block in DRAM. All OSD header blocks, as well as OSD bit maps, must start on 128-bit word address boundaries. The last OSD display section is denoted by an OSD header with its next header pointer set to "all ones".. There must be the same number (even) of OSD pixels in each line of a given OSD block. The first OSD pixel within an OSD block begins at the most significant bit of the first OSD byte.

During each frame that it is enabled, the OSD module requests memory accesses starting at the memory location pointed to by the OSD header pointer register (OHP). The memory controller services this request and fills the OSD FIFO with the first header block. At this point the OSD reads the header to determine where the bottom and top OSD field bit maps are located. The OSD first sets the appropriate address to the local memory controller, and then makes a request for the appropriate pixel data to the local memory controller. After receiving the pixel data, the OSD module waits for the display counters to reach the correct position, and then it outputs the OSD data. New memory accesses are requested as required to maintain the proper data flow through the OSD FIFO. When the last byte of pixel data for the current OSD region has been read into the OSD FIFO, the OSD sends a new header address to the local memory controller using the "next OSD header pointer". The OSD process is then repeated (read header, set data address, get pixel data, etc.) up through and including the last OSD region for the current frame.

This process (linked list) is illustrated graphically in the accompanying diagram. Note that the "OHP" (which points to the first OSD header block) is initialized via a Host Bus register. Thus, each header block has 3 pointers in it: a bottom pixel block pointer, a top pixel block pointer, and a next OSD header block pointer. This structure allows the user to physically separate (in memory) the location of the header blocks and data blocks, as well as allow the reuse of data blocks for different headers.



Example of the Linked List structure of the OSD

D3. OSD DATA FORMATS

D3.1. PALETTE-BASED DATA

The OSD pixels that are stored in memory are a 4:4:4 representation of chroma/luma levels set from palettes. Each palette entry contains a transparency bit, a blending bit, 6 bits of Y, 4 bits of Cb, and 4 bits of Cr. Before a color component is output, it is "left justified"(lower bits padded with zeros) to produce 8 bits each of Y, Cb, and Cr. The transparency bit and the blending bit for each entry allow the user to selectively display either an OSD pixel, a video pixel, or a blended pixel or a pixel by pixel basis. The blending (mix) weight for the current OSD block is defined in the header (Section D9).

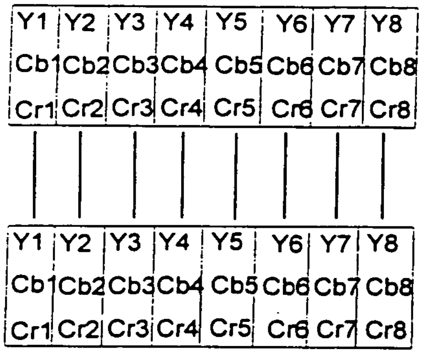
For YCbCr multiplexed eight-bit output, the 4:4:4 representations must be muxed into 4:2:2 MPEG video output. Thus, after "left justification", every other chroma pair must be dropped. This effectively reduces the OSD chroma resolution to be 1/2 of the luma resolution.

D3.2. TRUE COLOR BASED DATA

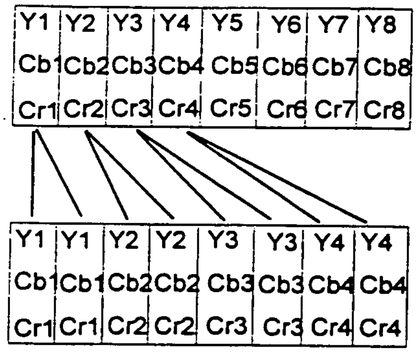
The OSD also supports a non-palette based display mode called "True Color". In this mode, the data coming into the OSD from its FIFO is assumed to be in 4:2:2 format. For this format, every four bytes of data represent 2 pixels (8 bits Y1, 8 bits Cb1, 8 bits Cr1, and 8 bits Y2). The same chroma components are used for both Y1 and Y2. This format is repeated for successive pixels, resulting in an effective pixel format of 16 bits per pixel.

D4. RESOLUTION MODES

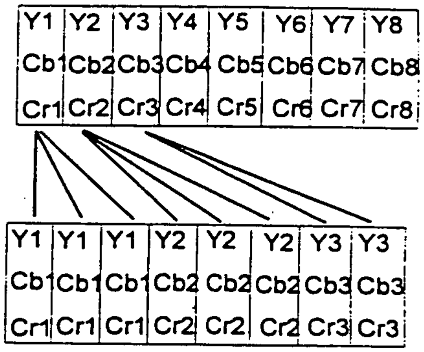
High resolution displays can have 4 to 6 times the pixel density for the same size OSD area. To ease setup requirements on the control microprocessor, half resolution and a one-third resolution modes are provided. These modes are available for both palette based OSD and true-color OSD. The following diagrams show how individual pixels are mapped to output pixels based upon the selected resolution. Note that one-third resolution mode is not supported for 8 bit output mode.


Y1 Y2 Y3 Y4 Y5 Y6 Y7 Y8
Cb1 Cb2 Cb3 Cb4 Cb5 Cb6 Cb7 Cb8
Cr1 Cr2 Cr3 Cr4 Cr5 Cr6 Cr7 Cr8
Y1 Y2 Y3 Y4 Y5 Y6 Y7 Y8
Cb1 Cb2 Cb3 Cb4 Cb5 Cb6 Cb7 Cb8
Cr1 Cr2 Cr3 Cr4 Cr5 Cr6 Cr7 Cr8

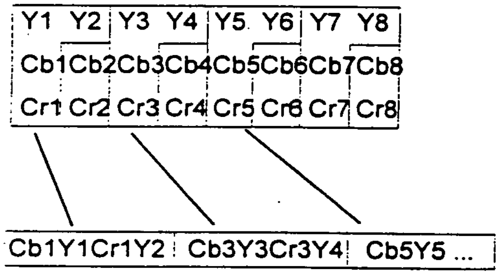
Full Resolution OSD, 24-bit


Y1 Y2 Y3 Y4 Y5 Y6 Y7 Y8
Cb1 Cb2 Cb3 Cb4 Cb5 Cb6 Cb7 Cb8
Cr1 Cr2 Cr3 Cr4 Cr5 Cr6 Cr7 Cr8
Y1 Y1 Y2 Y2 Y3 Y3 Y4 Y4
Cb1 Cb1 Cb2 Cb2 Cb3 Cb3 Cb4 Cb4
Cr1 Cr1 Cr2 Cr2 Cr3 Cr3 Cr4 Cr4

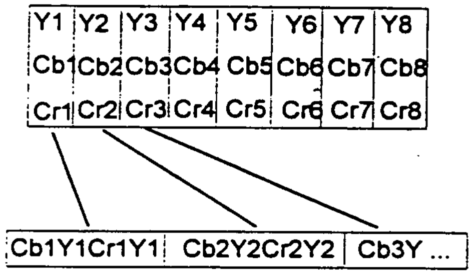
Half Resolution OSD, 24-bit


Y1 Y2 Y3 Y4 Y5 Y6 Y7 Y8
Cb1 Cb2 Cb3 Cb4 Cb5 Cb6 Cb7 Cb8
Cr1 Cr2 Cr3 Cr4 Cr5 Cr6 Cr7 Cr8
Y1 Y1 Y1 Y2 Y2 Y2 Y3 Y3
Cb1 Cb1 Cb1 Cb2 Cb2 Cb2 Cb3 Cb3
Cr1 Cr1 Cr1 Cr2 Cr2 Cr2 Cr3 Cr3

Third Resolution OSD, 24-bit


Y1 Y2 Y3 Y4 Y5 Y6 Y7 Y8
Cb1 Cb2 Cb3 Cb4 Cb5 Cb6 Cb7 Cb8
Cr1 Cr2 Cr3 Cr4 Cr5 Cr6 Cr7 Cr8
Cb1Y1Cr1Y2 Cb3Y3Cr3Y4 Cb5Y5 ...

Full Resolution OSD, 8-bit


Y1 Y2 Y3 Y4 Y5 Y6 Y7 Y8
Cb1 Cb2 Cb3 Cb4 Cb5 Cb6 Cb7 Cb8
Cr1 Cr2 Cr3 Cr4 Cr5 Cr6 Cr7 Cr8
Cb1Y1Cr1Y1 Cb2Y2Cr2Y2 Cb3Y ...

Half Resolution OSD, 8-bit

It is important to realize that it is up to the user to properly set the starting/ending column positions of the OSD based upon both the resolution mode and the output mode. If the OSD is in 24 bit output mode, then the starting and ending columns in the OSD header map directly correspond to output display positions (based upon the internal display pixel counter). For example, if the column start position is 100 and the end position is 199 (200 pixels), then the OSD will begin displaying pixels at pixel count 100 and end at pixel count 199. Thus, this function is not dependent upon the resolution mode selected. If the resolution is Half, and the user wants to display this same set of pixels, then he would have to adjust the ending column position to 399. The effect would be to see a horizontally "stretched" version of the original OSD picture.

For 8 bit mode, the column start and end positions map to output display positions differently than in the 24-bit mode. Essentially, the internal pixel counter is running twice as fast in this mode and the chroma and luma data is being multiplexed onto the luma output port. In this mode, in order to make the OSD appear at the same relative position as in 24 bit mode, the column start/end positions would need to be modified as follows:

(1) 8-bit mode column start position = 2* (24-bit column start position)
(2) 8-bit mode column end position = (number of OSD pixels in line) *2 + 8-bit mode column start position - 1

D5. OUTPUT MODES

The OSD supports five different output modes for both 8 bit and 24 bit output modes. In addition, the OSD allows the luma component of the output to be delayed with respect to the choma components while in 24 bit output mode. A 4-bit Host Bus value defines how many clock cycles the luma is to be delayed (0 to 15) for the associated chroma components.

Each output mode is selected based upon the active region of display, as well as OSD host bus registers, and the current OSD header, as described in the following paragraphs. Note that for any output mode, the OSD will limit the eight bit luma (Y) value of the output between 1 and 254.

D5.1. Blended OSD and Video

Blended OSD and video is the selected output format whenever the following conditions are satisfied:

(1) Video output is currently enabled
(2) The display is in an active OSD region
(3) Blending is enabled, transparency is disabled
(4) The OSD mix weight is not zero

Each palette entry has its own blending and transparency bits. However, the mix weight is defined in the header for the entire OSD region. For "True Color" mode, there is no blending available; however, transparency can be obtained by setting the luma value of a pixel to all "zeros."

D5.2. OSD (unblended)

OSD (unblended) is the selected output format whenever the following conditions are satisfied:

(1) Video output is currently enabled
(2) The display is in an active OSD region
(3) Blending is disabled, transparency is disabled

This is the "normal mode" of the OSD, in which OSD pixels simply replace video pixels.

D5.3. Fixed Video Level

A fixed Video Level is the selected output format whenever the following conditions are satisfied:

(1) Video output is currently enabled
(2) OSD is not being displayed (blended or unblended)
(3) Video Override is enabled by Host Bus register

This mode uses fixed video parameters from the Host Bus for the Y, Cr, and Cb values to output.

D5.4. Video (unblended).

Video (unblended) is the selected output format whenever the following conditions are satisfied:

(1) Video output is currently enabled
(2) OSD is not being displayed (blended or unblended)
(3) Video Override is disabled by Host Bus register

This is "normal" video output mode, when the OSD is disabled.

D5.5. Blanking Video

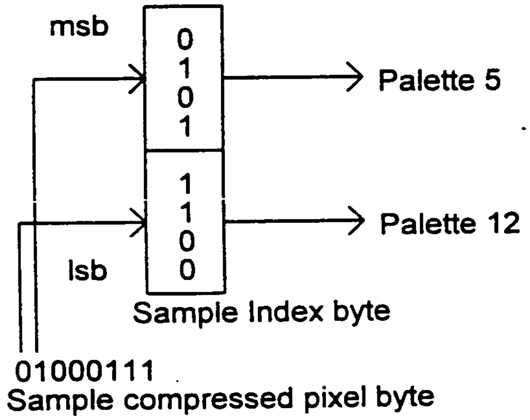
Blanking Video is the default output whenever the video output is disabled. The luma value is set to a Host Bus programmable black level, and each chroma component (Cr, Cb) is set to a blanking level of 128.

D6. COMPRESSED PIXEL MODE

An additional mode is supported which allows the user to specify "runs" of one bit pixels. The user supplies a bitstream consisting of an index byte, followed by bytes of pixels. Each byte of pixel data contains 8 one bit pixels. The index byte contains two 4 bit indexes. A run length counter tells the OSD how many one bit pixels there are in each run. Note that the compressed run length counter is independent of the resolution mode selected. For example, if the compressed run length counter equals 20, then the next three bytes following the index contains those twenty "one-bit" pixels, whether the OSD is in full, one half, or one third resolution models. However, the run length must be an even number, and be no less than four. At the end of each run, the following byte must be the next index byte, followed by the next run. All runs must be the same length, as specified by the 7-bit compressed run length count found in the header block. Furthermore, the user must also specify the bytes per line count in the header, taking into account both pixel bytes and the associated indexes to arrive at the correct total bytes per line. In the figure below, note that the msb of each compressed pixel byte corresponds to the 1st pixel out. If a bit is 1, the most significant four bits of the index are sent to address the palette. If a bit is 0, then the least significant four bits of the index are sent to address the palette. When enabled, this mode applies to all OSD pixels for that header block (all pixels are assumed to be compressed).

index0	byte0	byte1	byte2	......	index1	byte0	byte1	byte2

The above shows the structure of a sample compressed pixel bitstream.



The above shows how pixels in each byte reference the two selected palettes.

D7. VERTICAL LINE DOUBLING MODE

When this mode is enabled, every line of the OSD is repeated. Each line must begin on an 128-bit word address boundary. For example, by placing 5 lines of OSD data in memory, and setting the vertical enable bit to true, the OSD would produce 10 lines on the display output It is necessary, then, for the user to set the row start and end positions to correctly correspond to the total displayed OSD lines (in this case, 10).

D8. TRUE COLOR MODE

The "True Color" mode is enabled by setting the True Color Ena bit to "1" in the OSD header block. For this mode, the OSD will extract 4:2:2 data directly from the OSD FIFO, and send it out through the outputs, by-passing the normal palette look-up. All normal functions of the OSD are supported, except for the following:

(1) No mixing function is supported -
(2) The One-bit Compressed mode is not supported.

Thus, the user may use this mode in full, half, or third resolutions. Twenty-four and eight bit output modes are also supported. Any pixel can be made transparent by setting its Y component to "0". Note that all data for the header block is assumed to be in 4:2:2 format, with data retrieved by the OSD in the following sequence: Y1, Cb1, Cr1, Y2, Y3, Cb3, Cr3, Y4, ... Each component is 8 bits of data. In this format, both Y1 and Y2 use the same Chroma components (Cb1, Cr1). This results in an effective pixel size of 16 bits/pixel. As in palette-based mode, only even numbers of pixels are supported for OSD blocks. Again, the user needs to properly calculate the number of pixels per line required for the given resolution and line width.

D9. OSD HEADER DEFINITION

Each OSD block consists of a header which contains OSD positioning information, control logic registers, pointers to the next OSD header as well as the current BOTTOM and TOP data pointers, and the palette definition. The OSD is enabled by setting OSD ENA bit in the host bus interface. All OSD headers have a structure of header[0-31][15:0], where the first "[]" contains the word number within the OSD header, and the second "[]" contains the bits used

D9.1. Row Start/End Pos, Column Start/End Pos

The row start and end, and the column start and end pointers define an inclusive box in which OSD will be displayed. Outside of this box, no OSD will be displayed.

row_start_pos[10:0] = header[0][10:0]
row_end_pos[10:0] = header[1][10:0]
column_start_pos[11:0] = header[2][11:0]
column_end_pos[11:0] = header[3][11:0]

D9.2. True Color Ena

If True_Color_Ena = '1' then the OSD uses the incoming 4:2:2 data as 16 bit true color pixels (Chroma repeated for every two pixels).
If True_Color_Ena = '0' then the OSD assumes that incoming data references the OSD palette.

true_color_ena[0] = header[4][15]

D9.3. Use Existing Palette

To remove some of the throughput requirements of the micro, a bit is provided that instructs the OSD to use the previously loaded palette for the current header block. The user must have previously loaded the palette before setting this bit; otherwise, the output of the palette is undefined.
If Use_Existing_Palette = '1' then the OSD does not reload the palette for the current header.
If Use_Existing_Palette = '0' then the OSD does reload the palette from the current header block.

use_existing_palette[0] = header[4][14]

D9.4. Bytes Per Line

The bytes per line count tells the OSD how many bytes are in each line of the OSD. Each line of OSD begins on 128-bit word boundarys.

bytes_per_line[9:0] = header[4][11:0]

D9.5. Mode ("M") bit

The "M" or mode bit tells the OSD whether 4-bit pixels or 2-bit pixels are to be used. If 2-bit pixels are to be used, then only four colors can be addressed in the palette. This bit is ignored when the compressed_pix_ena bit ("C" bit) is true. When M=0, 2 bits/pixel is chosen. When M=1, 4 bits/pixel is chosen.

M[0] = header[5][15]

D9.6. Vertical Line Doubling ("V") bit

The "V" bit or vertical line doubling enable bit tells the OSD whether or not to double each line. When V=0, line doubling is disabled. When V=1, line doubling is enabled.

V[0] = header[5][14]

D9.7. Res Mode bits

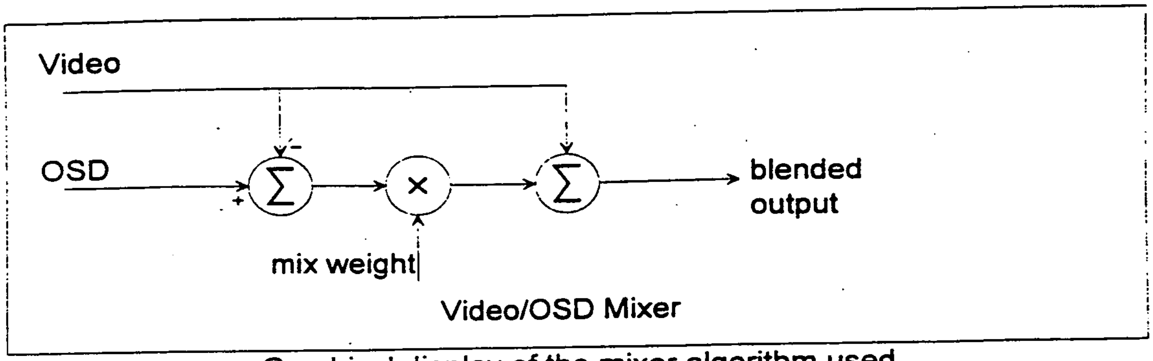
The res_mode bits tell the OSD the current resolutions: full, half, or third. This resolution is independent of the current video display mode.
res_mode = 00 for full resolution
res_mode = 01 for half resolution
es_mode = 10 for third resolution
res_mode = 11 is reserved

res_mode[1:0] = header[5][13:12]

D9.8. Mix Weight

The mix weight bits tell the OSD the blending ratio for OSD and video. Each bit has a resolution of 1/16. The blending ratio ranges from 0 (transparent) to 15/16 (pixel almost solid). Since the OSD CbCr values are only 4 bits, they are converted to 8-bit quantities by multiplying them by 16. The OSD Y value is multiplied by 4 to produce a 8-bit quantity. This conversion occurs before the OSD is mixed with video. The same mix weight is used for all pixels that have their respective blending bit set (B[n]=1). The mix weight is ignored for palettes with blending disabled, resulting in solid OSD.

mix_weight[3:0] = header[5][11:8]



Graphical display of the mixer algorithm used.

D9.9. Compressed Pixel Enable ("C") bit

The "C" bit (compressed_pixel_enable bit) tells the OSD whether all pixel bytes are to be treated as compressed pixel data. See section D6. When C=0, pixel compression is disabled. When C=1 pixel compression is enabled.

C[0] = header[5][7]

D9.10. Compressed Pixel Run Length

The Compressed Pixel Run Length count tells the OSD the length of each pexel run, for "compressed pixel mode". The value of this count can be "don't care" when compressed pixel mode is not used.

compressed_pixel_run_length[6:0] = header[5][6:0]

D9.11. Next OSD Header Ptr

The Next OSD Header Ptr tells the OSD where the next header is located. The last header of the current OSD field is defined to be a a header in which the next OSD header ptr is set to all "ones". After this last header with its associated data is processed and displayed, the OSD is disabled until the next vertical sync.
next_OSD_header_ptr[19:0] is composed of two words, next_OSD_header_ptr[19:16] and next_OSD_header_ptr[15:0].

next_OSD_header_ptr[19:16] = header[6][3:0]
next_OSD_header_ptr[15:0] = header[7][15:0]

D9.12. OSD Bottom Data Ptr

One of two pixel data pointers for the OSD. The OSD_bottom_data_ptr tells the OSD where the bottom pixel data is located. After each vertical, the OSD looks at the status of the top/not bottom field flag to determine which data pointer to send the local memory controller.
OSD_bottom_data_ptr[19:0] is composed of two words, OSD_bottom_data_ptr[19:16] and OSD_bottom_data_ptr[15:0].

OSD_bottom_data_ptr[19:16] = header[8][3:0]
OSD_bottom_data_ptr[15:0] = header[9][15:0]

D9.13. OSD Top Data Ptr

One of two pixel data pointers for the OSD. The OSD_top_data_ptr tells the OSD where the top pixel data is located. OSD_top_data_ptr[19:0] is composed of two words, OSD_top_data_ptr[19:16] and OSD_top_data_ptr[15:0].

OSD_top_data_ptr[19:16] = header[10][3:0]
OSD_top_data_ptr[15:0] = header[11][15:0]

D9.14. Palette Table

The palette table has 16 entries (palette[0] through palette15], inclusive) constructed as palette[n][Tn,Bn,paletteYn[5:0],palleteCBn[3:0],paletteCRn[3:0]].

palette[n][15:0] = header[n+16][15:0]

Tn=palette[n][15], where Tn=0 means transparency disabled, and Tn=1 means transparency enabled, for palette[n].
Bn=palette[n][14], where Bn=0 mean blending disabled, and Bn=1 means blending enabled, for palette[n].

D9.15. Unspecified Header Bits

Some header bits (for example, header[6][8]) and some header words (for example, header[12][15:0]) are left unspecified and may take on any value.

D10. OSD HOST BUS REGISTERS

See Appendix E for register addresses.

D10.1. OSD CNTRL (OSDCTRL[6:0])

Type: 7-bit Register (double buffered)

Description:

Bit 7:	Unused
Bits 6-3:	Defines the Luma Delay with respect to chroma for 24 bit output mode. "1111" = luma delay of 15 clock cycles. "0000" = luma delay of 0 clock cycles.
Bit 2:	Video Override Ena. If '1' selects video override components as a possible output video.
Bit 1:	Twentyfour not Eight Bit Mode. If '1', OSD output is 24 bits YCrCb. If '0', OSD output is 8 bits multiplexed YCrCb.
Bit 0:	OSD Enable bit. If '1', OSD is enabled. If '0', OSD is disabled.

D10.2. Y VIDEO OVERRIDE (YOVR[7:0])

Type: 8-bit Register (double buffered)

Description:
This register holds the luma component of the fixed video output selection.

D10.3. Cb VIDEO OVERRIDE (CbOVR[7:0])

Type: 8-bit Register (double buffered)

Description:
This register holds the one of the chroma components (Cb) of the fixed video output selection.

D10.4. Cr VIDEO OVERRIDE (CrOVR[7:0])

Type: 8-bit Register (double buffered)

Description:
This register holds the one of the chroma components (Cr) of the fixed video output selection.

D10.5. Y BLACK LEVEL (YBLACK[7:0])

Type: 8-bit Register (double buffered)

Description:
This register holds the luma (Y) component for the blanking level of the video output selection

HD MPEG VIDEO DECODER

APPENDIX E

REGISTER MAP

Thomson Consumer Electronics, Inc.
Indianapolis, Indiana, USA

Revision No. 2.3

THESE DRAWINGS AND SPECIFICATIONS ARE THE PROPERTY OF THOMSON CONSUMER ELECTRONICS, INC. AND SHALL NOT BE REPRODUCED OR COPIED OR USED AS THE BASIS FOR THE MANUFACTURE OR SALE OF APPARATUS OR DEVICES WITHOUT WRITTEN PERMISSION.

TCE PROPRIETARY AND CONFIDENTIAL

E1. OVERVIEW

In general, register addresses are grouped by functionality. This address range spans two register banks: addresses 0x000 through 0x0FF spans the standard working register addresses, and 0x100 through 0x1FF span test mode addresses.

Full address range "AREG" is specified by AREG[0-255][8:0]

The working bank "REG" is specified by REG[0-127][8:0] = AREG[0-127][8:0]

The test bank "TREG" is specified by TREG[0-127]=AREG[128-255][8:0]

This document shall specify addresses in the form of BASE + INDEX, where the BASE address is dedicated for a specific design block and INDEX refers to the specific offset address in the design block.

Many of the registers contained herein form part of larger register pointer structures to external SDRAM. The size of the register pointer structure is determined as follows:

a.) total byte address = 16,777,216 bytes (128M bits)
b.) total number of bit do address = 24
c.) most memory pointers have a minimum 4-bit address granularity, which results in 128-bit word address.
d.) in order to preserve simplified double word access by the host processor, the following general memory pointer structure is created:

Big endian format is used for applications bus addressing.

for type=mpointer:
mpointer[address][23:4] will be organized as:
REG[address+1][23:16], REG[address+2][15:8], REG[address+3][7:4], 0000b.
Upon a read REG[address][7:0] will return 00000000b and REG[address+3][3:0] will return a 0000b

for type=dpointer:
dpointer[address][23:0] will be organized as:
REG[address+1][23:16], REG[address+2][15:8], REG[address+3][7:0].
Upon a read REG[address][7:0] will return 00000000b

for type = byte:
byte[address][7:0] will be organized as REG[address][7:0], where address may occur on any byte address boundary

for type = word:
word[address][15:0]= REG[address][15:8], REG[address+1][7:0], where address==address&&11111110b

for type = dword:
dword[address][31:0] = word[address][31:16],word[address+2][15:0], where address==address&&11111100b

for type = qword:
qword[address][63:0] = dword[address][63:32], dword[address][31:0], where address==address&&11111000b

for type = dqword:
dqword[address][127:0]=dword[address][127:64], dword[address][63:0], where address==address&&11110000b

This section will not describe the semantic for the specified registers. Said semantics will be found in the individual appendices.

E2. REGISTER ADDRESSES

E2.1. Configuration, control and miscellaneous register list

Register	width	double-buff.	meaning
MCF	5	bit depdnt	memory config (compression modes)
MST8		no	memory setup (SGRAM, refresh period interval)
CCF	3	edge-trig	chip configuration, enable interfaces
CTL	7	edge-trig	control register
CMD	4	action W	command reg (launch actions) + QM selection
HRC	8	edge-trig	local memory read, write and block copy control
MAF128		no	host interface for local memory access fifo
BCWC	20	no	block copy word count
BCBR	.20	no	block copy repeat
HDF	8	read-only	header data fifo read
ITM	19	edge trig	interrupt mask
ITS	19	read only	interrupt status
STA	19	read only	status register
QMW	8	write only	quantization table loading
EPS	16	no	compression epsilon values
TST	3	edge-trig	test command register

E2.2. Decoding instruction registers

Register	width	double-buff.	meaning
TIS	7	VSync	task instruction
PFH	8	DSync	Picture F-Parameters Horizontal
PFV	8	DSync	Picture F-Parameters Vertical
PPR1	6	DSync	Picture Parameters
PPR2	6	DSync	Picture Parameters

E2.3. LMC register list

Register	width	double-buff.	meaning
DFP	16	VSync	Display frame buffer (256 bytes unit)
RFP	16	DSync	Reconstruction frame buffer (256 bytes unit)
FFP	16	DSync	Forward frame buffer (256 bytes unit)
BFP	16	DSync	Backward frame buffer (256 bytes unit)
BBG1	16	SoftReset	Bit buffer 1st zone start address (256 bytes unit)
BBG2	16	SoftReset	Bit buffer 2nd zone start address (256 bytes unit)
BBS1	16	SoftReset	Bit buffer 1st zone end address (256 bytes unit)
BBS2	16	SoftReset	Bit buffer 2nd zone end address (256 bytes unit)
BBL	16	read only	VLD Bit buffer level
BBT	16	edge-trig	Bit buffer threshold
HRP	20	no	mem read pointer
HWP	20	no	mem write pointer
BSDA	20	no	block copy source/dest address
DFS	14	DSync	Decoded frame size (in macroblocks)

DFW	8	Dsync	Decoded frame width (in macroblocks)
DCHP	15	Vsync	Display chroma frame buffer (256 bytes unit)
RCHP	15	Dsync	Reconstruction chroma frame buffer (same unit)
XFS	14	Vsync	Display frame size (in macroblocks)
XFW	8	Vsync	Display frame width (in macroblocks)
FCHP	15	Vsync	Forward chroma frame buffer (256 bytes unit)
BCHP	15	Vsync	Backward chroma frame buffer (256 bytes unit)
OHP	20	Vsync	OSD header start pointer address

E2.4. Display register list

Register	width	double-buff.	meaning
HDO	12	VSync	horizontal drive start (inclusive)
HDS	12	VSync	horizontal drive stop (inclusive)
XDO	12	VSync	horizontal display window start, value of PIX_CNT for first pixel of active video
XDS	12	VSync	horizontal display window stop, value of PIX_CNT for last pixel of active video
CLKLN	12	VSync	clocks per line, total number of display clocks per horizontal period
VDO	12	VSync	vertical drive start (inclusive), relative to HALF_LINE_CNT
VDS	12	VSync	vertical drive stop (inclusive)
YDO	11	VSync	vertical display window start, value of LINE_CNT for first line of active video
YDS	11	VSync	vertical display window stop, value of LINE_CNT for last line of active video
PANR	11	VSync	horizontal pan vector, integer part
SCAN	10	Sync	left justified vertical scan vector
LSOCSO	8	VSync	luma/chroma H-SRC initialization, 4 bits each of sub-pixel horizontal scan vector
HLFLN	12	Sync	number of half lines per vertical interval
BPPLN	7	VSync	number of (16 pixel wide) block pairs per line
ORLN0	10	VSync	output read line0 Start
LSR	10	VSync	upsampling ratio = 1024/LSR
LMULN0	8	VSync	LMU line0 start
DRINT	8	VSync	display refresh interval
VFCRAM	8	circular	VFC instructions, to VFC RAM
VFCCTL	5	VSync	VFC instruct. read bank select (1 bit), VFC instruct. loop back address (4 bits)
DRINCR	8	VSync	display refresh increment
HCTRL	2	Vsync	HSRC bypass control
DRST	1	Vsync	Issues reset to raster generator
ACCFRDIF	8	Vsync	accumulated frame difference
FMCTRL	2	VSync	film mode control
LMUCTRL	4	VSync	LMU control
FMLPF	8	Vsync	Film mode - Lines per field
HVMODE	3	Vsync	Ext_HV_d1_not_HV_DRIVE, RAS_GEN_RST_EXT_HV,H_V_DRIVE_IN
VFCOFS	4	Vsync	VFC startup offset
VFCCNT	8	Vsync	VFC max loopbacks per field

E2.5. OSD Register list

OSDCTRL	8	VSync	OSD control
YOVR	8	VSync	Y override
CbOVR	8	VSync	Cb override
CrOVR	8	VSync	Cr override
YBLACK	8	VSync	Y blanking level

E2.6. Clock/PLL register list

Register	width	double-buff.	meaning

CLK_SEL.0	8	no	Clk0 and S_Clk0 select
CLK_SEL.1	8	no	Clk1 and S_Clk1 select
CLK_SEL.2	8	no	Clk2 and S_Clk2 select
CLK_SEL.3	8	no	Clk3 and S_Clk3 select
CKG_PLL.0	8	no	PLL0 M and N dividers
CKG_PLL.1	8	no	PLL1 M and N dividers
CKG_SEL.1	3	no	PLL1 source select

E2.7. Status Bits

- [21] HSRC_Yfifo_underflow
- [20] HSRC_Cfifo_underflow
- [19] OSD_fifo_underflow
- [18] Bitstream fifo full
- [17] New Discarded Packet (PES)
- [16] Inconsistency Error in PES Parser (PES)
- [15] New SCR latched (PES)
- [14] Decoding Overflow error
- [13] Decoding Underflow error
- [12] Decoding Semantic error (pipe error)
- [11] HRC_write not ready
- [10] HRC_read not ready
- [9] Block copy Idle
- [8] Start code detector fifo empty
- [7] Start code detector fifo nearly full
- [6] Pipeline idle
- [5] DSync
- [4] VSync top
- [3] VSync bottom
- [2] Bit buffer empty
- [1] Bit buffer (nearly) full
- [0] Header hit

E3. WORKING REGISTER MAP

00	MCF	20	DFS[13:8]	40	res PES	60	HDF
01	CCF	21	DFS[7:0]	41	res PES	61	CMD
02	CTL	22	DCHP[14:8]	42	res PES	62	res gen
03	TIS	23	DCHP[7:0]	43	res PES	63	res gen
04	PFH	24	RCHP[14:8]	44	PES_SC1[31:24]	64	res gen
05	PFV	25	RCHP[7:0]	45	PES_SC2[23:16]	65	res gen
06	PPR1	26	DFW	46	PES_SC3[15:8]	66	res gen
07	PPR2	27	XFW	47	PES_SC4[7:0]	67	res gen
08	HRC	28	XFS[13:8]	48	PES_CFG[6:0]	68	CDcntdum
09	QMW	29	XFS[7:0]	49	res PES	69	CDcnt[23:16]
0A	MST	2A	FCHP[14:8]	4A	VID_DSM[7:0]	6A	CDcnt[15:8]
0B	res gen	2B	FCHP[7:0]	4B	VID_TS5[3:0]	6B	CDcnt[7:0]
0C	DFP[15:8]	2C	BCHP[14:8]	4C	VID_TS1[31:24]	6C	SCDcntdum
0D	DFP[7:0]	2D	BCHP[7:0]	4D	VID_TS2[23:16]	6D	SCDcnt{23:16]
0E	RFP[15:8]	2E	res gen	4E	VID_TS3[15:8]	6E	SCDcnt[15:8]
0F	RFP[7:0]	2F	res gen	4F	VID_TS4[7:0]	6F	SCDcnt[7:0]
10	FFP[15:8]	30	PLL_DIV3[31:24]	50	PLL_DIV0[31:24]	70	ITSdum
11	FFP[7:0]	31	PLL_DIV3[23:16]	51	PLL_DIV0[23:16]	71	ITS[17:16]
12	BFP[15:8]	32	PLL_DIV3[15:8]	52	PLL_DIV0[15:8]	72	ITS[15:8]
13	BFP[7:0]	33	PLL_DIV3[7:0]	53	PLL_DIV0[7:0]	73	ITS[7:0]
14	BBG1[15:8]	34	PLL_DIV4[31:24]	54	PLL_DIV1[31:24]	74	STAdum
15	BBG1[7:0]	35	PLL_DIV4[23:16]	55	PLL_DIV1[23:16]	75	STA[17:16]
16	BBG2[15:8]	36	PLL_DIV4[15:8]	56	PLL_DIV1[15:8]	76	STA[15:8]
17	BBG2[7:0]	37	PLL_DIV4[7:0]	57	PLL_DIV1[7:0]	77	STA[7:0]
18	BBL[15:8]	38	PLL_DIV5[31:24]	58	PLL_DIV2[31:24]	78	ITMdum
19	BBL[7:0]	39	PLL_DIV5[23:16]	59	PLL_DIV2[23:16]	79	ITM[17:16]
1A	BBS1[15:8]	3A	PLL_DIV5[15:8]	5A	PLL_DIV2[15:8]	7A	ITM[15:8]
1B	BBS1[7:0]	3B	PLL_DIV5[7:0]	5B	PLL_DIV2[7:0]	7B	ITM[7:0]
1C	BBS2[15:8]	3C	CKG_PLL0[7:0]	5C	CLK_SEL0[7:0]	7C	EPS[15:8]
1D	BBS2[7:0]	3D	CKG_PLL1[7:0]	5D	CLK_SEL1[7:0]	7D	EPS[7:0]
1E	BBT[15:8]	3E	CKG_SEL1[7:0]	5E	CLK_SEL2[7:0]	7E	
1F	BBT[7:0]	3F	TST[2:0]	5F	CLK_SEL3[7:0]	7F	VERSION

80	HRPdum	A0	D1CTRL[1:0]	C0	HDO[11:8]	E0	display reserved
81	HRP[23:16]	A1	D1LNCNT[7:0]	C1	HDO[7:0]	E1	VFCCNT[7:0]
82	HRP[15:8]	A2	D1L1[9:8]	C2	HDS[11:8]	E2	LSR[9:8]
83	HRP[7:4]	A3	D1L1[7:0]	C3	HDS[7:0]	E3	LSR[7:0]
84	HWPdum	A4	D1L2[10:8]	C4	XDO[11:8]	E4	LMULN0[7:0]
85	HWP[23:16]	A5	D1L2[7:0]	C5	XDO[7:0]	E5	DRINT[7:0]
86	HWP[15:8]	A6		C6	XDS[11:8]	E6	VFCRAM[7:0]
87	HWP[7:4]	A7		C7	XDS[7:0]	E7	VFCCTL[6:0]
88	BCDRdum	A8		C8	VDO[11:8]	E8	DRINCR[7:0]
89	BCBR[23:16]	A9		C9	VDO[7:0]	E9	HCTRL[1:0]
8A	BCBR[15:8]	AA		CA	VDS[11:8]	EA	DRST[0]
8B	BCBR[7:4]	AB		CB	VDS[7:0]	EB	OSDCTRL[6:0]
8C	BCWCdum	AC		CC	YDO[10:8]	EC	YOVR[7:0]
8D	BCWC[23:16]	AD		CD	YDO[7:0]	ED	CbOVR[7:0]
8E	BCWC[15:8]	AE		CE	YDS[10:8]	EE	CrOVR[7:0]
8F	BCWC[7:4]	AF		CF	YDS[7:0]	EF	YBLACK[7:0]
90	OHPdum	B0	MAF[127:120]	D0	PAN[10:8]	F0	ACCFRDIF[7:0]
91	OHP[23:16]	B1	MAF[119:112]	D1	PAN[7:0]	F1	FMCTRL[1:0]
92	OHP[15:8]	B2	MAF[111:104]	D2	SCANR[14:8]	F2	LMUCTRL[3:0]
93	OHP[7:4]	B3	MAF[103:96]	D3	SCANR[7:5]	F3	FMLPF[7:0]
94	BSDAdum	B4	MAF[95:88]	D4	CLKLN[11:8]	F4	HVMODE[2:0]
95	BSDA[23:16]	B5	MAF[87:80]	D5	CLKLN[7:0]	F5	VFCOFS[3:0]
96	BSDA[15:8]	B6	MAF[79:72]	D6	LSOCSO[7:0]	F6	
97	BSDA[7:0]	B7	MAF[71:64]	D7	display reserved	F7	
98		B8	MAF[63:56]	D8	HLFLN[11:8]	F8	
99		B9	MAF[55:48]	D9	HLFLN[7:0]	F9	
9A		BA	MAF[47:40]	DA	BPPLN[6:0]	FA	
9B		BB	MAF[39:32]	DB	WRLN0[9:2]	FB	
9C		BC	MAF[31:24]	DC	ORLN0[9:8]	FC	
9D		BD	MAF[23:16]	DD	ORLN0[7:0]	FD	
9E		BE	MAF[15:8]	DE	display reserved	FE	
9F		BF	MAF[7:0]	DF	display reserved	FF	

E4. TEST REGISTER MAP

100	VLD bsh	120	LMC-req	140		160	PES_STATE[7:0]
101	VLD bsh	121	LMC-req	141		161	PES_PLENGTH[15:8]
102	VLD bsh	122	LMC-req	142		162	PES_PLENGTH[7:0]
103	VLD bsh	123	LMC-req	143		163	PES_HLENGTH[7:0]
104	VLD bsh	124	LMCrom	144		164	PES_LOCK1[7:0]
105	VLD bsh	125	LMCrom	145		165	PES_LOCK2[3:0]
106	VLD bsh	126	LMCrom	146		166	PES_LOCK3[7:0]
107		127	LMCrom	147		167	PES_LOCK4[4:0]
108		128	LMCrom	148		168	PES_LOCK5[5:0]
109		129		149		169	SCD-PES
10A		12A	P2-IQ	14A		16A	
10B		12B	P1-IQ	14B		16B	
10C		12C		14C		16C	
10D		12D		14D		16D	MVunit
10E	P1-OSP2	12E	LMCrom	14E		16E	
10F	P1-OSP2	12F	LMCrom	14F		16F	
110	P1-IQout	130	LMCamu	150		170	
111	P1-IQout	131	LMCamu	151		171	
112		132		152		172	
113		133		153		173	
114	P1-IDCT	134		154		174	
115	P1-IDCT	135		155		175	
116		136		156		176	
117		137		157		177	
118		138		158		178	
119		139		159		179	
11A		13A		15A		17A	
11B		13B		15B		17B	
11C		13C		15C		17C	
11D		13D		15D	MVunit	17D	
11E		13E		15E	MVunit	17E	
11F		13F	TST[2:0]	15F	MVunit	17F	

180	DEBUG[7:0]	1A0	YBTLRAM[2:0]	1C0	D1TSTCTRL[2:0]	1E0	
181	HSRC[1:0]	1A1		1C1	D1TSTDATA[7:0]	1E1	
182		1A2		1C2	D1TSTSTAT[2:0]	1E2	
183		1A3		1C3		1E3	
184		1A4		1C4		1E4	
185		1A5		1C5		1E5	
186		1A6		1C6		1E6	
187		1A7		1C7		1E7	
188	LDecomp[1:0]	1A8	CBTLRAM[2:0]	1C8		1E8	
189		1A9		1C9		1E9	
18A		1AA		1CA		1EA	
18B		1AB		1CB		1EB	
18C		1AC		1CC		1EC	
18D		1AD		1CD		1ED	
18E		1AE		1CE		1EE	
18F		1AF		1CF		1EF	
190	CDecomp[1:0]	1B0		1D0		1F0	
191		1B1		1D1		1F1	
192		1B2		1D2		1F2	
193		1B3		1D3		1F3	
194		1B4		1D4		1F4	
195		1B5		1D5		1F5	
196		1B6		1D6		1F6	
197		1B7		1D7		1F7	
198		1B8		1D8		1F8	
199		1B9		1D9		1F9	
19A		1BA		1DA		1FA	
19B		1BB		1DB		1FB	
19C		1BC		1DC		1FC	
19D		1BD		1DD		1FD	
19E		1BE		1DE		1FE	
19F		1BF		1DF		1FF	

E4.1. Configuration, control and miscellaneous register list

MCF[4:0]

Memory Configuration

Address: 00	Register Type: R/W
Data Type: Byte	Double Buffer: Bit Dependent
Reset: 0	

Composition:
MCF[5] : launch mode register set
MCF[4] : display progressive picture
MCF[3:2] : display decompression mode
(00 = no comp, 01 = 2M/3, 11 = H/2-M/2)
Double buffered on VSync
MCF[1:0] : decoding decompression mode
(same as above). Double buffered on DSync.

Description:
Configures memory compression modes

CCF[2:0]

Chip Configuration

Address: 01	Register Type: R/W
Data Type: Byte	Double Buffer: Edge Trig.
Reset: 0	

Composition:
CCF[2] : PBO or DMA (prevent bit buffer overflow) : stop access to bit buffer when full
CCF[1] : EDI : enable DRAM interface
CCF[0] : EVI : enable video interface

Description:
Configures chip, external interfaces

CTL[6:0]

Control Register

Address: 02
Data Type: Byte
Reset: 0

Register Type: R/W
Double Buffer: Edge Trig.

Composition:
CTL[6] : SPI : slice picture ID use for error concealment
CTL[5] : DEC : disable error concealment
CTL[4] : CFB : circular frame buffer
CTL[3] : ERU : automatic pipeline reset on underflow error
CTL[2] : ERO : automatic pipeline reset on overflow error
CTL[1] : ERS : automatic pipeline reset on semantic (pipe) error
CTL[0] : EDC : enable decoding

Description:
Control Register

MST[7:0]

Memory Setup

Address: 0A
Data Type: Byte
Reset: 0

Register Type: R/W
Double Buffer: Vsync

Composition:
MST[7] = use SGRAM not SDRAM memory chips
MST[6:0] = RFI[6:0] refresh period interval

Refresh period interval = 16xRFIxTmemClk
~ 35.2 x RFI x TsdCLk

For standard SDRAM with 15625 ns refresh period interval, and with 100 MHz sdClk : RFI = 44.

Description:
Select SDRAM/SGRAM type and set refresh interval

HD MPEG VIDEO DECODER

APPENDIX G

MEMORY REDUCTION

Thomson Consumer Electronics, Inc.
Indianapolis, Indiana, USA

Revision No. 2.1

TCE PROPRIETARY AND CONFIDENTIAL

TABLE OF CONTENTS

FIGURES

\BLES

G1. TOP LEVEL DESCRIPTION

G1.1. Overview

Not all applications of the HD-MPEG IC will involve full HD resolution display devices. Some low cost applications of the HD-MPEG IC can tolerate less than perfect MPEG decoding.

The HD-MPEG IC must provide two modes of reduced memory operation which include:
1) anchor frame compression or
2) anchor frame compression + in loop horizontal detail reduction.

Normal decoding of HD images requires 96 Mbits of external memory. Method 1 above allows virtually indistinguishable decoding of HD images with only 64 Mbits of external memory or slightly reduced quality decoding with only 48 Mbits. Method 2 above allows medium definition decoding of HD images with only 32 Mbits of external memory.

The processing techniques required in the HD-MPEG IC for memory reduction modes of operation are d escribed in the following sections.

The following figure shows conceptually the location of the required memory reduction processes within the MPEG decoding module.

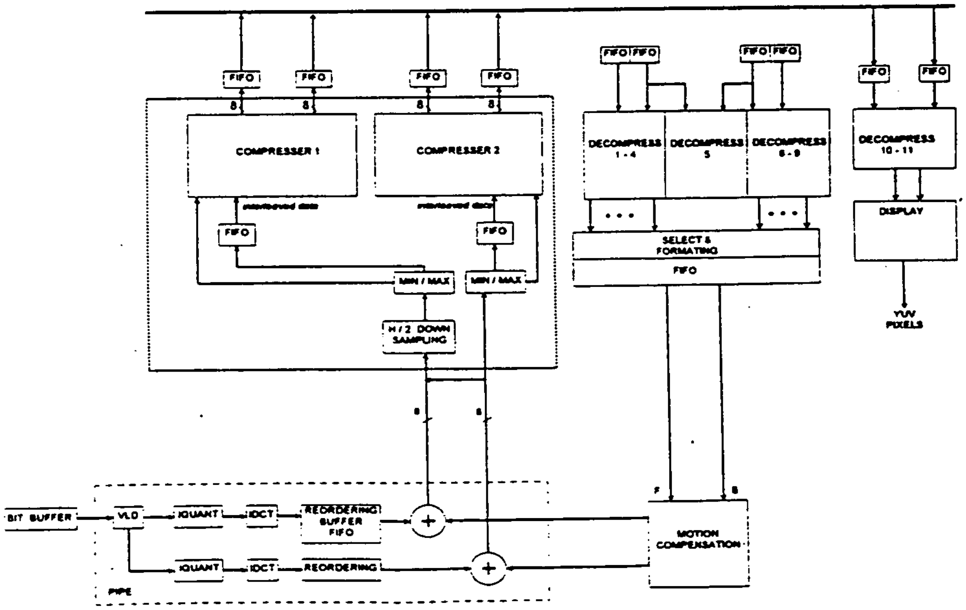


Figure G1 - MPEG Decode Loop with Memory Reduction

As shown in the preceding figure, the memory reduction processes lie between the external frame memory and the rest of the decoder processing.

Two independent concepts are used for memory reduction.

Block based compression/decompression is a nearly lossless compression process which reduces the amount of storage needed to hold 8x8 luma blocks or 4x4 U/V blocks. The amount of reduction is selectable and can be 25% or 50%. Low contrast/detail blocks are usually losslessly compressed while high contrast/detail blocks usually experience some loss especially in the 50% reduction case. Even though the decompressed pixel blocks may have differences from the original pixels, block based compression/decompression is the first choice for memory reduction over horizontal detail reduction and produces the least amount of decoder loop drift relative to the ideal decoder.

Horizontal detail reduction reduces the external memory requirements by reducing the number of pixels which are stored in the memory. This technique uses a horizontal spatial lowpass filter followed by a 2:1 horizontal decimation ahead of the memory write process. The full resolution image is formed after reading from memory by simple pixel replication. This technique is lossy and causes a drift in the MPEG decoding loop (as compared to the encoder loop); it is only used when block based compression/decompression is used.

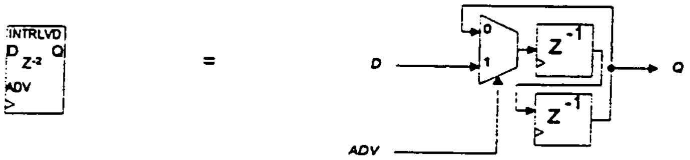


Figure G2 - Interleaved flip-flop used in diagrams

G1.2. Register List

OPERATIONAL BUS REGISTERS				
Section	Name	# bits	Address	Note
	H2ENA	1	G1	
H2	??	7	G1	spare
COMPRESS *	BLKCMPENA	1	G2	
COMPRESS *	CMPMODE	1	G2	M/2 OR 2M/3.
COMPRESS	??	6	G2	spare
DCMPRS_PRED *	BLKCMPENA	1	G3	
DCMPRS_PRED *	CMPMODE	1	G3	M/2 OR 2M/3.
DCMPRS_PRED	??	6	G3	spare
DCMPRS_DSPL *	BLKCMPENA	1	G4	
DCMPRS_DSPL *	CMPMODE	1	G4	M/2 OR 2M/3.
DCMPRS_DSPL	??	6	G4	spare

* These bits are used in the compression section, the predictor decompression, and the display decompression. It is assumed that it is preferable to repeat these controls two times then to route these nets to the three different sections.

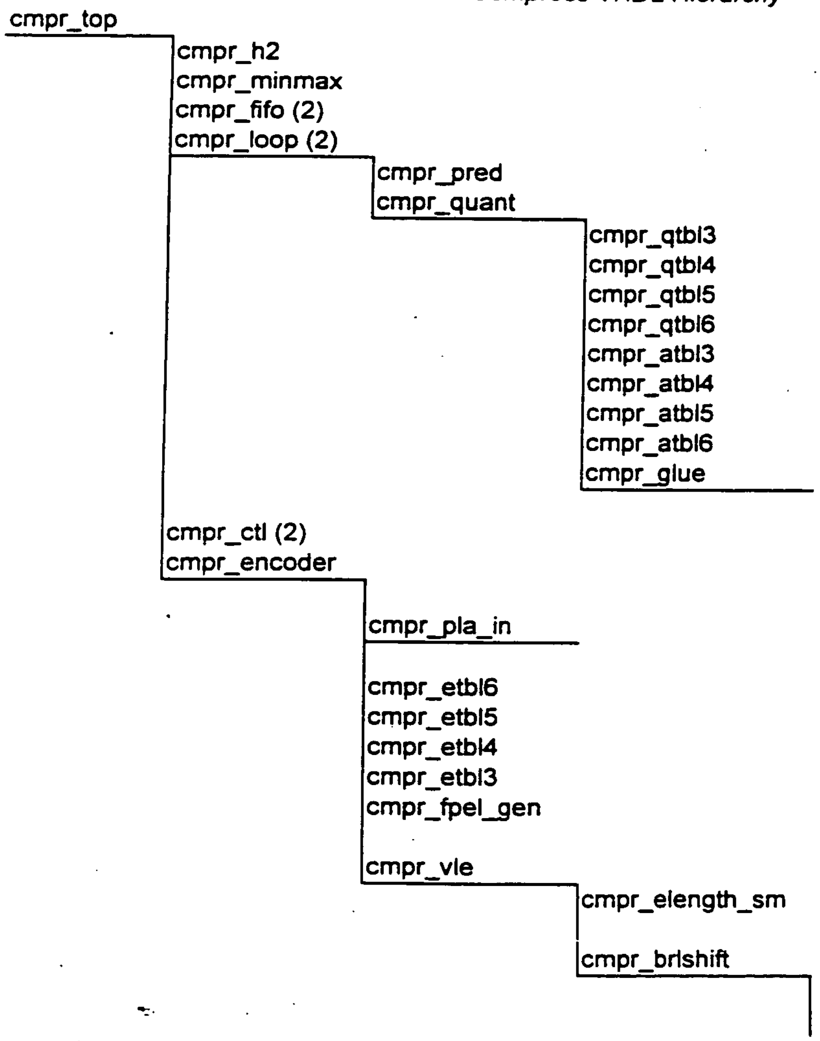
TEST BUS REGISTERS

Section	Name	# bits	Address	Note
H2	H2ENA	1	GT1	
COMPRESS		8	GT2	misc test controls
COMPRESS		8	GT3	misc test read register
BLK_COMPRESS		8	GT5	misc test read register
"		8	GT6	misc test controls
"		8	GT7	misc test controls
"		1	GT8	Predictor test ctl
"		7	GT8	
DCMPRS_PRED		8	GT9	misc test read register
"		4	GT10	selects dcmp # for G9
"		4	GT11	selects internal pt for G9
"		8	GT12	spare
DCMPRS_DSPL		8	GT13	misc test read register
"		4	GT14	selects dcmp # for G9
"		4	GT15	selects internal pt for G9
"		8	GT16	spare

G2. COMPRESSION

G2.1. Compression Section

Table G1- Compress VHDL Hierarchy



G2.1.1. Inputs

IN_PIPE1(7:0):
Eight bit sample provided by the 1st pipe.

IN_PIPE2(7:0):
Eight bit sample provided by the 2nd pipe.

ACK1:
Acknowledge signal from the adder block 1.

ACK2:
Acknowledge signal from the adder block 2.

BLKSTART1:
This active high bit indicates that the current 32 word on IN_PIPE corresponds to the start of an 8x8 luma or 4x4 chroma block. This signal can be a one clock wide pulsed synchronis with the clock *TBD*.

BLKTYPE1:
This bit indicates whether the data on IN_PIPE corresponds to a luma or chroma block. This signal need be valid only while BLKSTART is high. A '0' indicated the data corresponds to an 8x8 luma block, and a '1' indicated a 4x4 chroma block.

BLKSTART2:
This active high bit indicates that the current 32 word on IN_PIPE corresponds to the start of an 8x8 luma or 4x4 chroma block. This signal can be a one clock wide pulsed synchronis with the clock *TBD*.

BLKTYPE2:
This bit indicates whether the data on IN_PIPE corresponds to a luma or chroma block. This signal need be valid only while BLKSTART is high. A '0' indicated the data corresponds to an 8x8 luma block, and a '1' indicated a 4x4 chroma block.

G2.1.2. Outputs

COMP_OUT1a(7:0):
Compressed data ready to be stored in Imc FIFO.

COMP_OUT1b(7:0):
Compressed data ready to be stored in Imc FIFO.

COMP_OUT2a(7:0):
Compressed data ready to be stored in Imc FIFO.

COMP_OUT2b(7:0):
Compressed data ready to be stored in Imc FIFO.

REQ1:
This active high bit indicates that the compressors are ready for data from Pipe1.

REQ2:
This active high bit indicates that the compressors are ready for data from Pipe2.

G2.1.3. Bus Registers

address G1 *bit* (0) H2ENA:
This active high bit enables the H/2 section.

G1(1) CMPMODE:
This register controls the compression mode.

Test Bus Write Addresses:

G2(0)T_INPUT_SEL :
Controls the data source for the two inputs to compression. '0' (default) selects the pipe outputs, '1' selects the read FIFOs from Decompression Section.

G2(1:4)T_FIFO_LD :
Forces the compression input FIFOs to load data on subsequent clocks.

G2(5)T_FORCE_START :
Forces the START signal to the compression blocks.

G2(6)T_FORCE_BLKTYPE :
Forces the BLKTYPE signal to the compression blocks.

Test Bus Read Addresses:

G3(2:0) T_RANGE:
Monitors selected lines from min/max block.

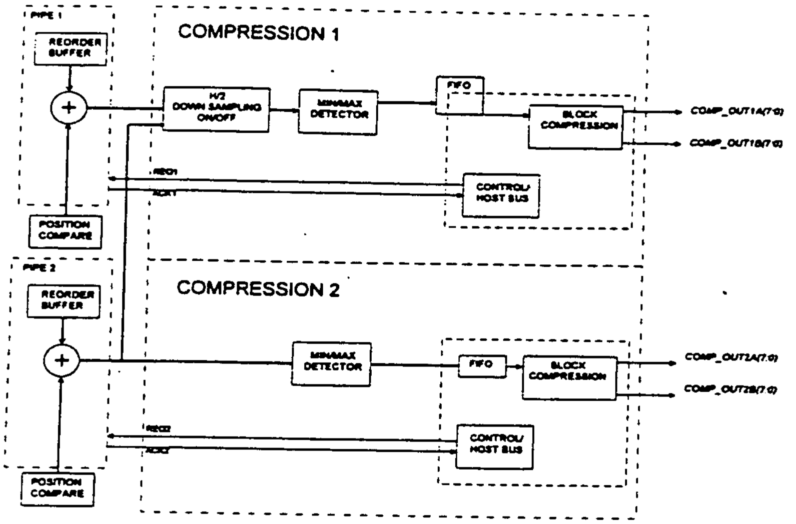


Figure G3- Top Level Memory Compression

G2.1.4. Description

The pipe sources data to the compression section via two 8 bit 54 MHz buses IN_PIPE1 and IN_PIPE2. Pipe 1 will output data from 2 luma and 1 chroma blocks of a given macroblock while PIPE2 will source data from the other blocks in that macroblock. There is no guarantee of the data from the two pipes being synchronized with each other except when in H/2M/2 mode. The average data rate of this interface will be < 108 Msamples/sec, where a sample corresponds to one 8 bit luma or chroma sample.

Data received from the pipe passes through the H/2 section, which decimates the data horizontally by a factor of 2. From H/2, data is passed to the Min/Max Scan Section. The block compression algorithm requires the range of the data being compressed and the minimum value to properly choose the appropriate quantization curve.

The two compression blocks will compress a given sample in two clock cycles. In order to increase through-put, data from two blocks of data will be interleaved such that a given compressor will compress two blocks in the same time as a compressor that can compress a sample each clock.

G2.2. Horizontal Detail Reduction

G2.2.1.Overview

Horizontal detail reduction involves horizontally lowpass filtering the decoded pixel data followed by 2:1 decimation in the horizontal direction before the pixels are written to the memory. When read from the memory to support motion compensation, the pixels are up sampled 2:1 in the horizontal direction by simple pixel duplication in the motion compensation unit. When read from the memory for display, the sample duplication will be performed in display section.

The lowpass filter used is a 2 tap symmetric FIR type with the following coefficients.

C0	C1	scalar
1	1	2

Note this filter must operate in the horizontal spatial domain and does not involve filtering across block boundaries.

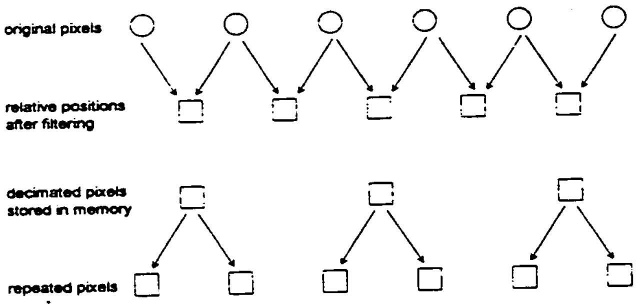

original pixels
relative positions after filtering
decimated pixels stored in memory
repeated pixels

Figure G4- Horizontal Filtering/Decimation/Up Sampling Process

As shown in figure ?, the 2 tap filter has the effect of shifting the relative position of the output pixels by 1/2 sample period relative to the input (with some integer period offset). The pixel replication used for up sampling has the effect of maintaining the same spatial position of the down sampled/up sampled pixels relative to the original pixels.

G2.2.2. Inputs

H2_IN(7:0):
Eight bit sample from the pipe. 64 conecutive words constitute a luma block and 16 consecutive words constitute a chroma block.

H2ENA:
This active high bit enables the H/2 section.

G2.2.3. Outputs

H2_OUT(7:0):

When this section is enabled, this represents a decimated pixel. When this H/2 is disabled, these bits are connected to *H2_IN(7:0)*.

G2.2.4. Bus Registers

N/A

G2.2.5. Description

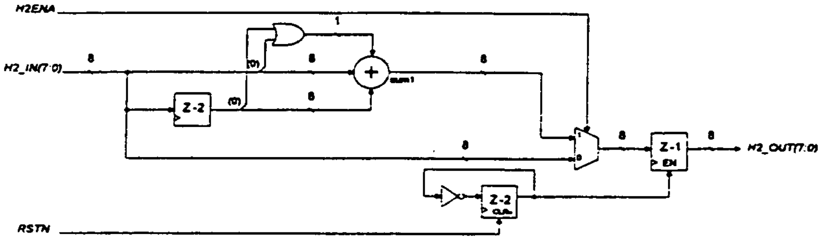


Figure G5- H/2 Section

G2.3. Min/max Scan

G2.3.1. Overview

The compression algorithm chooses the quantization table based on the range of the data present in the block of data. This block finds the min and max and generates a discrete range selection.

G2.3.2. Inputs

D_IN(7:0):
Input data to be scanned.
START:
This control bit clears the stored minimum and maximum value. This should be asserted sychronously with the start of a new block of data.

G2.3.3. Outputs

D_OUT(7:0):
This bus is D_IN() delayed by 1 clock.
MIN(7:0):
Eight bit minum value for the incoming luma/chroma block
RANGE(2:0):
Three bit value that represents the range of data in the just scanned block.

G2.3.4. Description

This block scans the incoming block of data to find the minimum and maximum values. When START goes high, the minimum and maximum registers are cleared and, depending on the state of BLKTYPE, the next 4 or 16 32 bits words are scanned.

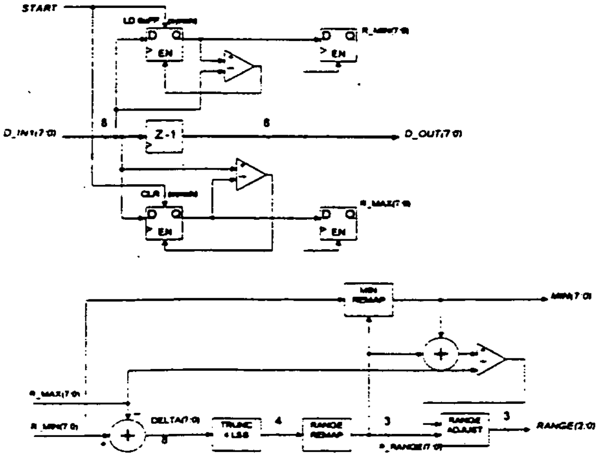


Figure G6- Min Max Scan Section

G2.4. Block based compression

G2.4.1. Subdesigns:

Compression Ctl
Compression Input Buffer
Compressor
Encoder

G2.4.2. Overview

The block based compression operates on 8x8 luma field blocks and 4x4 U or V blocks. The compression achieved is as follows:

Block Size	Uncompressed	25% Reduced	50% Reduced
8x8 blocks	512 bits	384 bits	256 bits
4x4 blocks	128 bits	64 bits	64 bits

Each field based block is scanned in a raster manner as shown in the following figure.

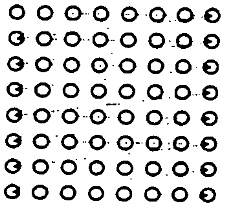


Figure G7 - Scan Path for Block Compression

The compression is achieved using a DPCM loop with adaptive prediction and non-linear fixed quantizing of the differences. Two passes are used to determine the minimum and maximum pixel values within the block; this ormation is used to adaptively select between non-linear quantizing rules to be used for the block.

necessary to interleave the compression hardware such that it takes two clock cycles to calculate each pixel. To maximize throughput and minimize area, the data from two independent blocks is interleaved on alternate clocks as it is fed into the compression sections. At the output of each of the two compression sections, compressed data is alternately read from the two output ports.

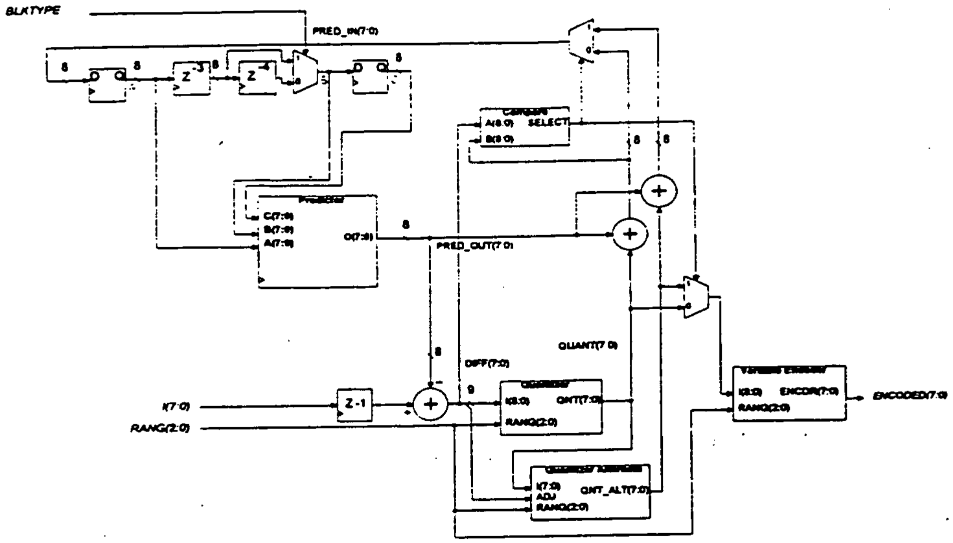


Figure G8 - Compression Function

G2.4.3. Inputs

D_IN1(7:0):
Eight bit data from Min/Max Section1.

D_IN2(7:0):
Eight bit data from Min/Max Section2.

RESETN:
Active low asynch reset

BLKTYPE:
This bit indicates the block type. '0' indicates an 8x8 luma block while '1' indicates a 4x4 chroma block.

START:
This control bit indicates the data at DATAIN corresponds to the start of a block of data. Note that this signal must be delayed from the START received from the pipe to compensate for the H/2 block.

COMP_ENA:
This control bit enables the compression section

MODE:
This control bit indicates the current compression mode: 2M/3 or M/2H/2.

LMC_REQ1:
.This active high signal indicates the LMC FIFO is ready for data from ComprOut1.

LMC_REQ2:
This active high signal indicates the LMC FIFO is ready for data from ComprOut2.

G2.4.4. Outputs

ComprOut1(7:0):
Compressed eight bit data in field macroblock format.

ComprOut2(7:0):
Compressed eight bit data.

LMC_ACK1:
This active high signal acknowledges to the LMC FIFO that ComprOut1 data is valid.
LMC_ACK2:
This active high signal acknowledges to the LMC FIFO that ComprOut2 data is valid.

G2.4.5. Bus Registers

G4(1:0) CompressMode:
This two bit register selects the compression mode:
00- no compression
01- 2M/3 compression
11- H/2M/2 compression

G 2.4.6.Test Bus Registers

G 2.4.6.1. Read Addresses:
G5(7:0) TestData:
This eight bit read register can be connected to various points of the compression block.

G 2.4.6.2. Write Addresses:
G6(?:0) TestDataSel:
This regsiter selects the test points available through the TestData port:
00 - Predictor output
01- Encoder output
02- Quantized Encoder outputG7(x) Predictor Disable:
This bit disables the predictor block

G2:4.7. Description

Data is stored in the two input FIFOs as the two Min/Max Sections scan the blocks.

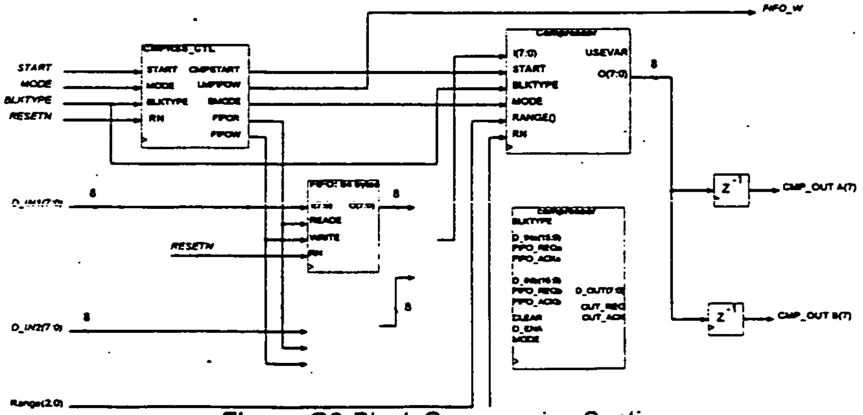


Figure G9-Block Compression Section

The 8 bit words received from the two Min/max Scan sections are placed in the two input FIFOs. The compression controller then reads out one byte at a time alternately from each FIFO. The compressed data is written 8 bits at a time to the LM FIFOs.

Figure G10-Compression Timing

The compression sections are always clocking data. They are reset at the start of a block, and the valid results are clocked into their respective output buffer. When compression is disabled, the entire block is bypassed via a mux.

Each block of compressed data begins with overhead bits to indicate the parameters necessary for decompression. Three types of information are included: a three bit index that represents the range of the data, a quantized representation of the minimum value of the block, and a representation of the first pixel. In most cases, this representation is the first pixel value minus the block minimum truncated. The overhead bits are defined below in table 8.

Table G2- Overhead Bits

Range	0	1	2	3	4	5	6	7	8	9	10	11	12	13	comment	bits lost to overhead
16	0	0	0	D_3	D_2	D_1	M_7	M_6	M_5	M_4	M_3	M_2	M_1	M_0	Δ & mimum value	10
32	0	1	0	D_4	D_3	D_2	D_1	I_2	I_1	I_0					Δ & mimum index	6
64	0	1	1	D_5	D_4	D_3	D_2	D_1	I_2	I_1	I_0				Δ & mimum index	7
96	1	0	0	D_6	D_5	D_4	D_3	D_2	D_1	I_2	I_1	I_0			Δ & mimum index	8
128	1	1	0	D_6	D_5	D_4	D_3	D_2	D_1	I_2	I_1	I_0			Δ & mimum index	8
192	1	0	1	D_6	D_5	D_4	D_3	D_2	D_1	I_2	I_1	I_0			Δ & mimum index	9
256	1	1	1	P_7	P_6	P_5	P_4	P_3	P_2	P_1					first pel trunc to 7	6
	↑ range ↑			↑ First Byte ↑					↑ Second Byte ↑							

	range bits
M	minimum value
I	index into Minimum Lookup Table
P	first pel, quantized
D	first pel - min, unless noted otherwise

G2.4.8. Compression Controller

G2.4.8.1. Overview

G2.4.8.2. Inputs

MODE:

This bit indicates the current compression mode: '0' for 2M/3, '1' for H/2M/2. Note that in the 2M/3 mode, chroma blocks are compressed by 50% and luma blocks by 25%.

BLKTYPE:

This bit indicates the block type. '0' indicates an 8x8 luma block while '1' indicates a 4x4 chroma block.

START:

This control bit indicates the data at DATAIN corresponds to the start of a block of data.

Rn:

Active low asynchronous reset

G2.4.8.3. Outputs

CMPSTART:

This control bit resets the variable and fixed compression sections. This is a delayed version of the input START to compensate for the delay through the Min/Max Scan section.

BMODE:

This control bit selects the appropriate quantization and Huffman tables for either 2M/3 or H/2M/2 mode. A '0' selects the 2M/3 tables and the '1' selects the H/2M/2 tables.

FIFO_W:

This signal clocks the 32 bit data into the Compression Input Section

FIFO_R:

This signal clocks the 8 bit data out of the Compression Input Section

LMFIFO_W:

This signal clocks the 32 bit data into the local memory FIFO

G2.4.8.4. Description

The compression control block synchronizes all enables and resets to the block compression . It is also ʾsponsible for table selects and block type controls. Its specific tasks are:

- control the generation and insertion of overhead bits into the compressed data
- control the writing of data into the two sets of two Input FIFOs
- alternately clocking 8 bit data out of the two Compression Input Buffers to each Compressor
- selecting the 2M/3 or H/2M/2 quantizer tables. In 2M/3 mode, luma blocks are compressed 25% while chroma blocks are compressed 50%.

G2.4.9. Compressor

G2.4.9.1. Subdesigns:
Predictor
Fixed Encoder

G2.4.9.2. Overview

The compressor is a standard DPCM loop using the predictor and adaptive quantizer describe in subsequent sections.

G2.4.9.3. Inputs

D_In(7:0):
Eight bit interleaved data to be compressed.
RANGE(2:0):
Three bit range select from Min Max Scan. .
RESETN:
Active low asynch reset
BLKTYPE:
This bit indicates the block type. '0' indicates an 8x8 luma block while '1' indicates a 4x4 chroma block.
START:
This control bit indicates the data at DATAIN corresponds to the start of a block of data. Note that this signal must be delayed from the START received from the pipe to compensate for the H/2 and Min/Max sections.
MODE:
This control bit indicates the current compression mode: 0 selects 2M/3 and 1 selects H/2M/2 mode.

G2.4.9.4. Outputs

ENCODED(7:0):
Eight bit encoded data to be compressed.

G2.4.9.5. Description

In order to minimize the width of the barrel shifters and the number of clocks required to strip overhead bits out of the bit stream, the first byte of data stripped off before the barrel shifter. This allows a 6 bit barrel shifter to handle the remaining overhead bits in one additional clock cycle.

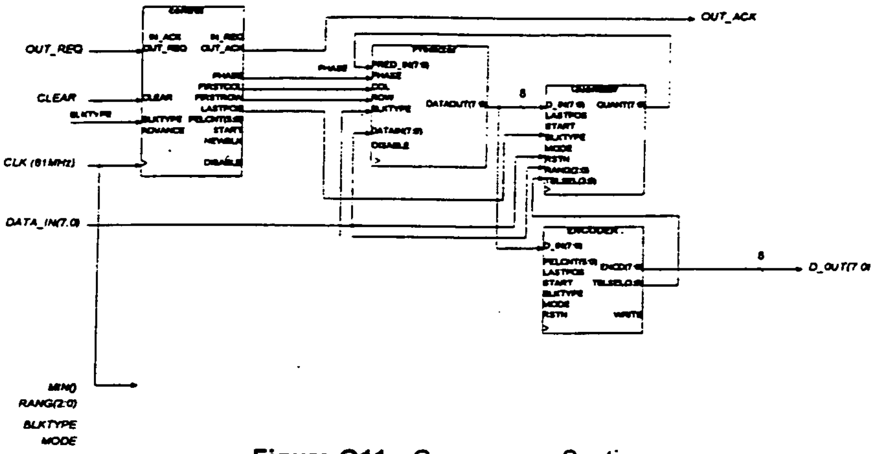


Figure G11- *Compressor Section*

G2.4.9.5.1. Predictor

G2.4.9.6. Overview

The same adaptive prediction technique is used in the compression and decompression sections. The pixel to be coded is predicted by previously coded pixels (since they are known to the decompression section). The following figure shows the pixels used to make the prediction.

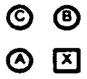


Figure G12- *Predictor Pixel Relationship*

In the figure X is the next pixel to be coded. A, B, and C are previously coded pixels known to the decompression circuit. A prediction of X is made using A, B, and C. The following pseudo code describes the algorithm which must be used:

```
if        $(|A-C|<e_1 \; \&\& \; |B-C| > e_2)$        $X_{pred} = B;$
else if   $(|B-C|<e_1 \; \&\& \; |A-C| > e_2)$        $X_{pred} = A;$
else                                          $X_{pred} = (A+B)/2;$
```

Note the above algorithm is only valid for pixels not in the first row or first column of the block. The exceptions are handled as follows:

1) The first pixel of the block is not coded
2) The pixels of the first row use A as the predictor
3) The pixels of the first column use B as the predictor

The same algorithm is used in both the fixed and variable paths.

G2.4.9.7. Inputs

DATAIN(7:0):
Eight bit quantized value of the current pixel.

COL:
This active high bit indicates the current pixel is located in the first column of a luma/chroma block.

ROW:
This active high bit indicates the current pixel is located in the first row of a luma/chroma block.

BLKTYPE:
This control bit indicates the block type. '0' indicates an 8x8 luma block while '1' indicates a 4x4 chroma block.

DISABLE:
This active high control bit causes the delay line to stop advancing data. The delay line does continue to alternate the phase of the current interleaved block.

G2.4.9.8. Outputs

DATAOUT(7:0):
Eight bit decompressed pixel value.

G2.4.9.9. Bus Registers

G8(0) PredEnable:
This active high bit enables the predictor. When low, the predictor output is 0 and the registers are cleared to 0.

G2.4.9.10. Description

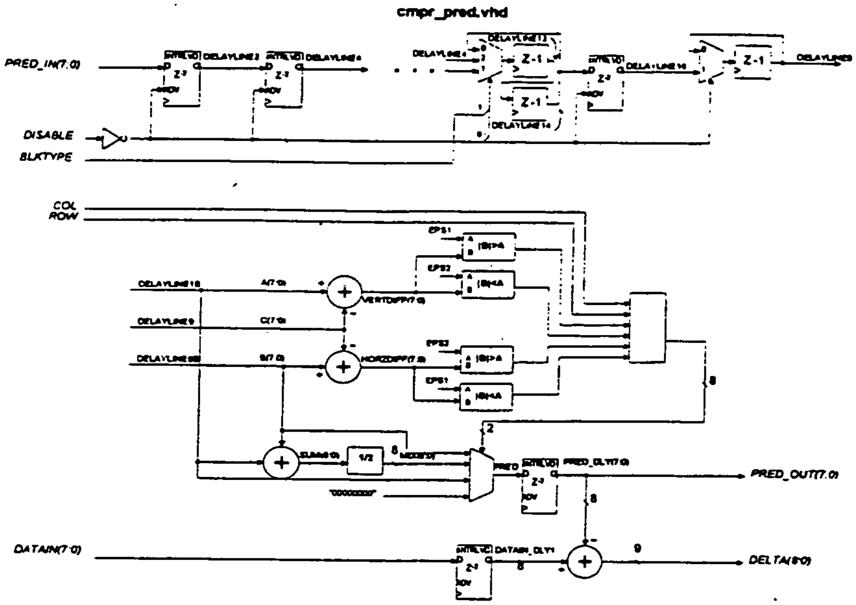


Figure G13- Compression Predictor Block

G2.4.10. Encoder

G2.4.10.1. Overview

The compressor requires a quantizer in the loop to match the quantizer/inverse quantizer operation that will occur on data received at the decompressor. The output of the compressor must also be encoded using the adaptive quantization method.

G2.4.10.2. Inputs

D_IN(7:0):

Eight bit difference data from subtractor.

MODE:

This control bit indicates the current compression mode: 0 selects 2M/3 and 1 selects H/2M/2 mode.

G2.4.10.3. Outputs

ENCODED(7:0):

Eight bit encoded data for output. On alternate clocks, this byte will represent data from two different blocks of data.

RANGE(2:0):

Three bit range select from Min Max Scan. .

QUANT(7:0):

Eight bit quantized data for the compressor loop. On alternate clocks, this byte will represent data from two different blocks of data.

G2.4.10.4. Description

This block produces two outputs: an eight bit quantized value for the compressor loop and a variable width encoded pixel that will be sent to the local memory. This block also generates the overhead bits for each compressed data block.

The encoding scheme is given in *TBD*.

Table G3- Encoding Tables

3 Bit Table

Level #	Index
7	001
6	011
5	101
4	111
3	100
2	010
1	000
0	110

4 Bit Table

Level #	Index
13	0011*
12	0101
11	0111
10	1001
9	1011
8	1101
7	1111
6	1100
5	1010
4	1000
3	0110
2	0100
1	0000*
0	1110

5 Bit Table

Level #	Index
25	00111*
24	01001
23	01011
22	01101
21	01111
20	10001
19	10011
18	10101
17	10111
16	11001
15	11011
14	11101
13	11111
12	11100
11	11010
10	11000
9	10110
8	10100
7	10010
6	10000
5	01110
4	01100
3	01010
2	01000
1	00000*
0	11110

Level #	Index
63	000001
62	000011
61	000101
60	000111
59	001001
58	001011
57	001101
56	001111
55	010001
54	010011
53	010101
52	010111
51	011001
50	011011
49	011101
48	011111
47	100001
46	100011
45	100101
44	100111
43	101001
42	101011
41	101101
40	101111
39	110001
38	110011
37	110101
36	110111
35	111001
34	111011
33	111101
32	111111

Level #	In
31	11
30	11
29	11
28	11
27	11
26	11
25	11
24	11
23	10
22	10
21	10
20	10
19	10
18	10
17	10
16	10
15	01
14	01
13	01
12	01
11	01
10	01
9	01
8	01
7	00
6	00
5	00
4	00
3	00
2	00
1	00
0	0

- Short Code Words. The indicated bits are arbitrary and chose to facilitate the symmetric tables.

The quantize-to-encode and encode-to-quantize tables are symmetric; therefore, only half of the tables are needed in the IC. An output remapping block generates the top half of the table from the lower half. The lsb of the input is used to separate the top and bottom half of the table.

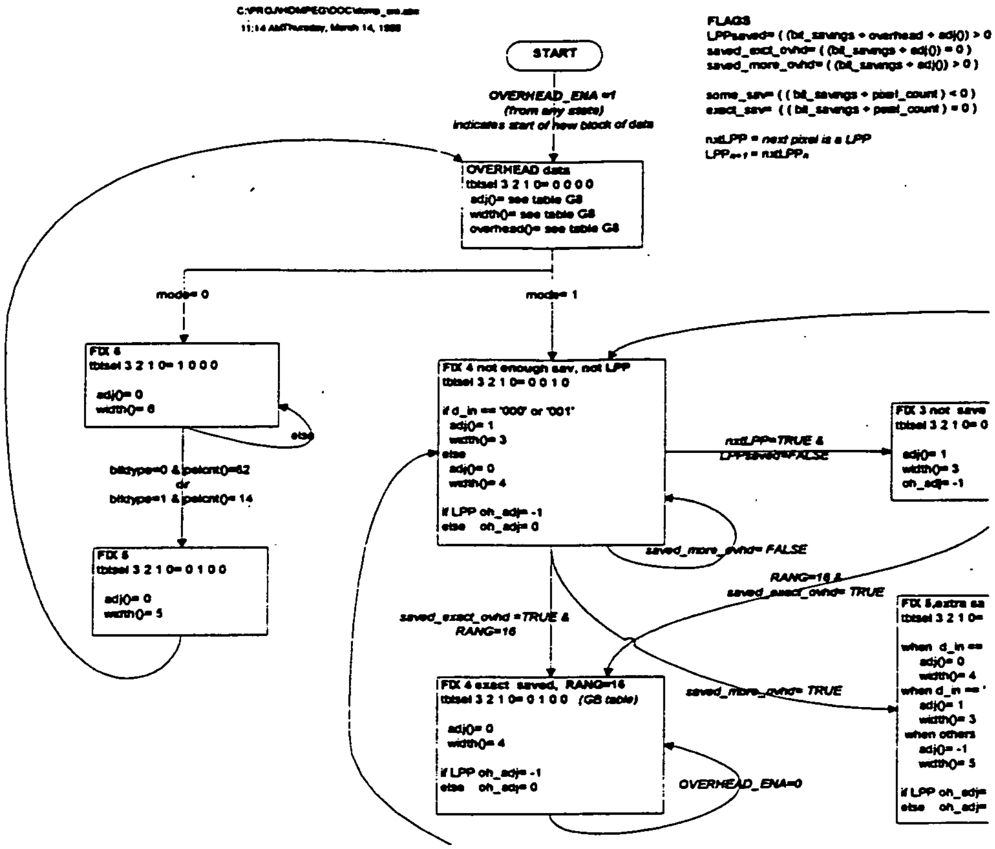


Figure G14- Encoder Length State Machine

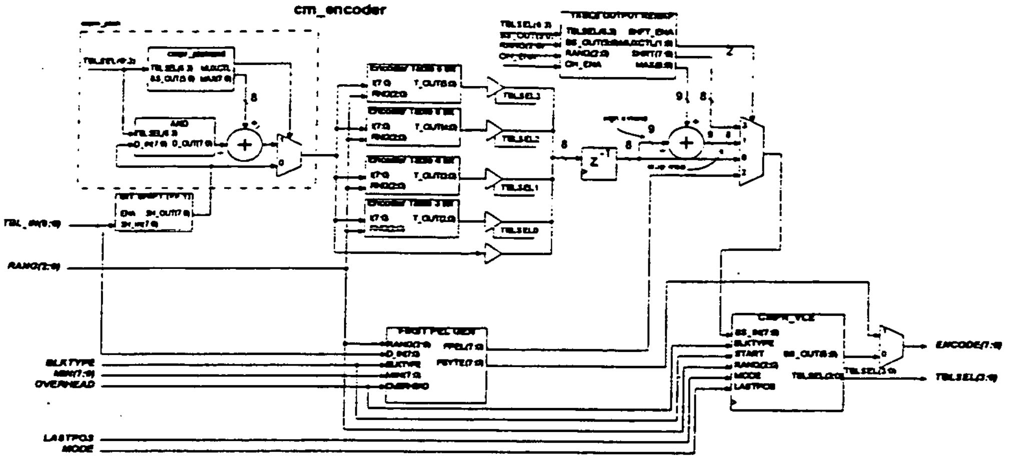
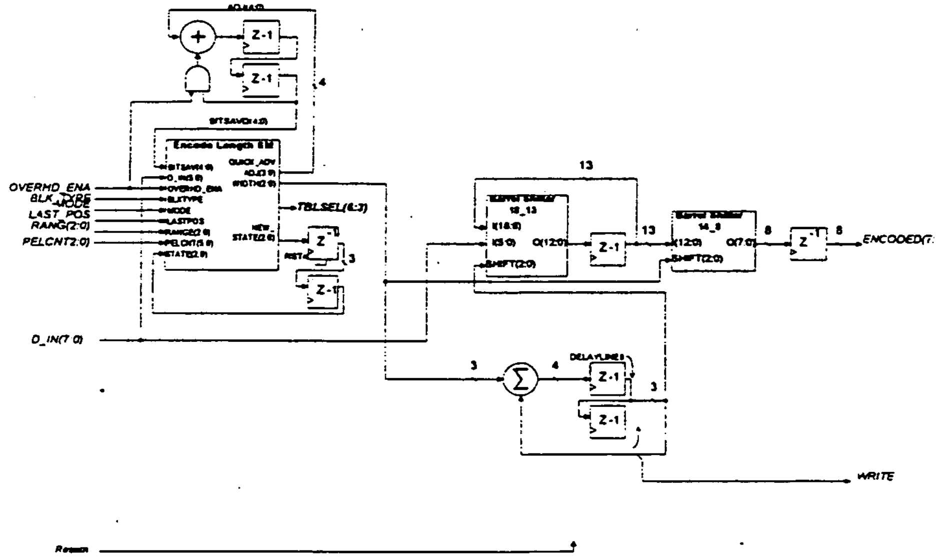


Figure G15- Variable Length Encoder Section



THOMSON CONSUMER ELECTRONICS CONFIDENTIAL AND PROPRIETARY

Figure G16- Encoder Barrel Shifter Section

The following tables are included in the encoding section.

		Range bits	Input bits	Output Bits	Symmetric
Quantization	cmpr_qtbl3	2	2+9	8	No
	cmpr_qtbl4			8	No
	cmpr_qtbl5			8	No
	cmpr_qtbl6			8	No
Alternate Table	cmpr_atbl3	2	2+91	8	
	cmpr_atbl4			8	
	cmpr_atbl5			8	
	cmpr_atbl6			8	
Encoding table	cmpr_etbl3	2		3	
	cmpr_etbl4			4	
	cmpr_etbl5			5	
	cmpr_etbl6			6	

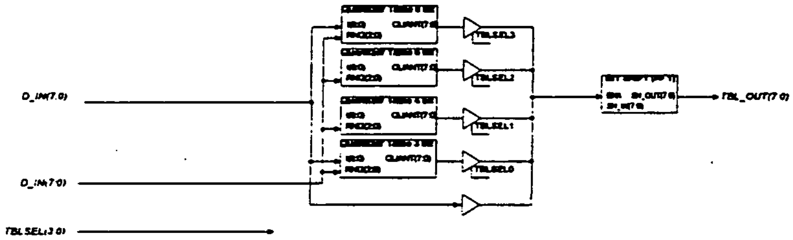


Figure G17- Encoder Section

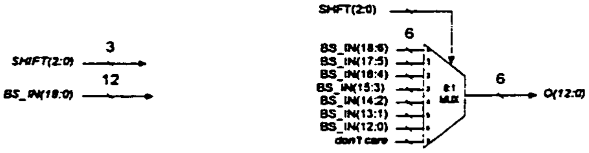


Figure G18- Barrel Shifter 19 input 13 output

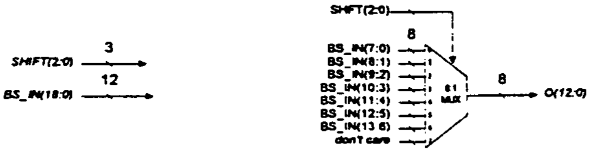


Figure G19- Barrel Shifter 13 input 8 output

G3. DECOMPRESSION

G3.1. Overview

The HDMPEG IC contains two decompression sections: the predictor decompression section and the display decompression section. These sections are between the local memory and the Motion Compensation Unit and between the local memory and the display section. The data is decompressed as it is fetched from the memory.

There are a total of 11 decompression blocks. Nine are for predictor decompression and two are for display decompression. For the predictor decompression, the local memory FIFOs provide two independent blocks of compressed data to each decompressor 32 bits at a time. Each decompressor provides a 32 bit output plus acknowledge to the motion compensation unit. -

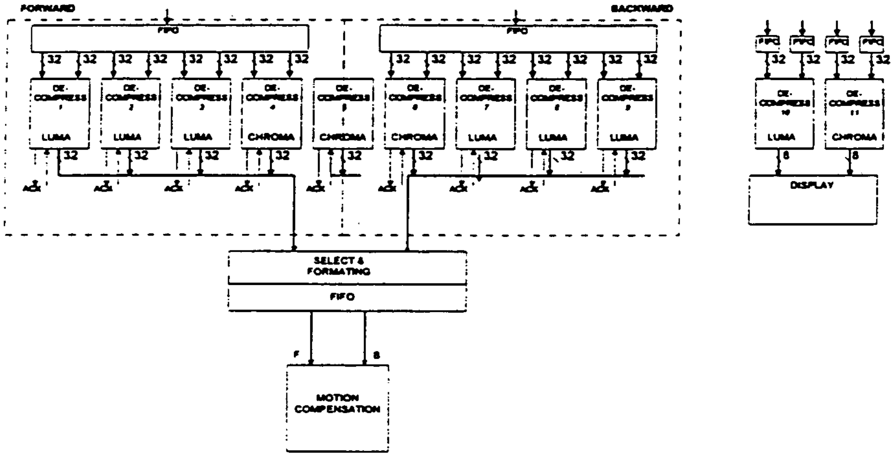


Figure G20- Top Level Decompression

Table G4- Decompress VHDL Hierarchy

- dcmp_top
  - dcmp_pred
  - dcmp_hshake
  - dcmp_ctl
  - dcmp_decoder
    - dcmp_pla_in
    - dcmp_pla_out
      - dcmp_tridrv8
    - dcmp_dtbl6
    - dcmp_dtbl5
    - dcmp_dtbl4
    - dcmp_dtbl3
    - dcmp_fpel_gen
    - dcmp_rng_gen
    - dcmp_vld
      - dcmp_dlength_sm
      - dcmp_dlength_ctl
      - dcmp_brlshift
        - dcmp_trimux2
        - dcmp_trimux8

G3.2. Predictor Decompression Top Level

G3.2.1. Inputs

rbus(20:0):
Internal communication bus used to control operational and debug modes and monitor status registers.

G3.3. Display Decompression Top Level

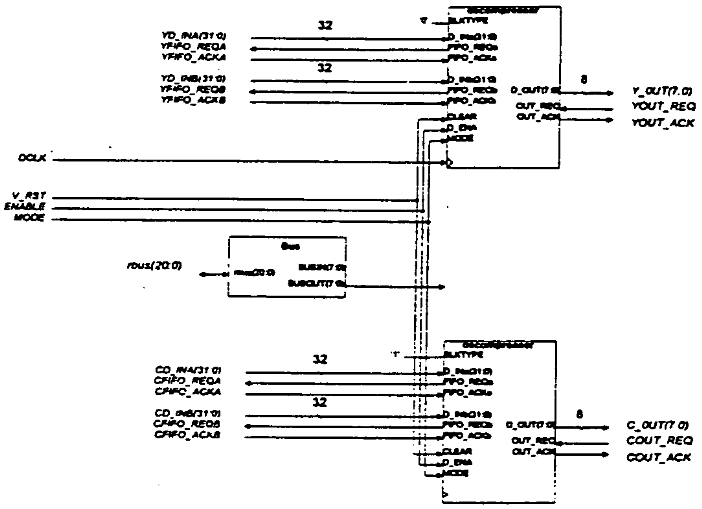


***Figure G21**- Display Decompression*

3.3.1. Inputs

OUT_REQ:
This active high signal from the display section indciates it is ready for new data.

ENABLE:
This active high bit enables the decompression section. When low, this section is bypassed and the GDCLK is disabled.

rbus(20:0):
Internal communication bus used to control operational and debug modes and monitor status registers.

G3.3.2. Outputs

D_OUT(7:0):
Uncompressed luma data for the display section.
OUT_ACK:
This active high signal indciates the data on D_OUT is valid.

G3.3.3. Bus Registers

G3.3.3.1. R Bus Addresses:
xx(1:0) CompressMode:
This two bit register selects the compression mode:
00- no compression
01- 2M/3 compression
11- M/2 compression
10- Illegal mode

G3.3.3.2. Test Bus Read Addresses:
G9(7:0) TestData:
This eight bit read register can be connected to various points of the compression block.
01- Selects the pixel counter from the contol block
02- Selects the valid data counter from the control block

G3.3.3.3.Test Bus Write Addresses:
G10(7:0) TestData:
This eight bit register selects the test points connected to bus register G9().

G3.4. Block based Decompressor

G3.4.1. Subdesigns:

- Predictor
- Decompression Controller
- Decompression Handshake Controller
- Decoder

G3.4.2. Description

Two independent blocks of data are provided to a decompressor using two Request/Acknowledge handshake lines. The reset, VSYNC for display decompress and DSYNC for predictor decompress asynchronously resets the entire section. After the reset is cleared, the barrel shifter shifts data through eight bits at a time. Data flow is from the two FIFOs to the handshake controller block. This block interleaves the 32 bit inputs from the FIFOs and provides it to the barrel shifter as it is needed. The first 16 bits of a block are used on the first clock cycle. or the rest of the block, it is provided eight bits at a time. When the first data word is ready at the output of the barrel shifter, the data valid signal goes active. If the data desitnation 's request is active, acknowledge goes high and operation continues. If no request is present, the decompressor stops until the next request.

The interface between the handshake controller and the barrel shifter is via DATA READY/ACK. If data is not availalbe from the FIFOS, the barrelshifter will clock the data in anyway by ADVANCE going active. The decompress controller will notice the ADVANCE went high with DATAREADY inactive, indicating that invalid data was clocked in the shifter. The remaining data will be shifted out of the shifter until the end of the next block, since it is guanteed that the FIFO will never go empty in the middle of a compressed blcok of data. Once the current block is clocked out of the shifter, the handshake

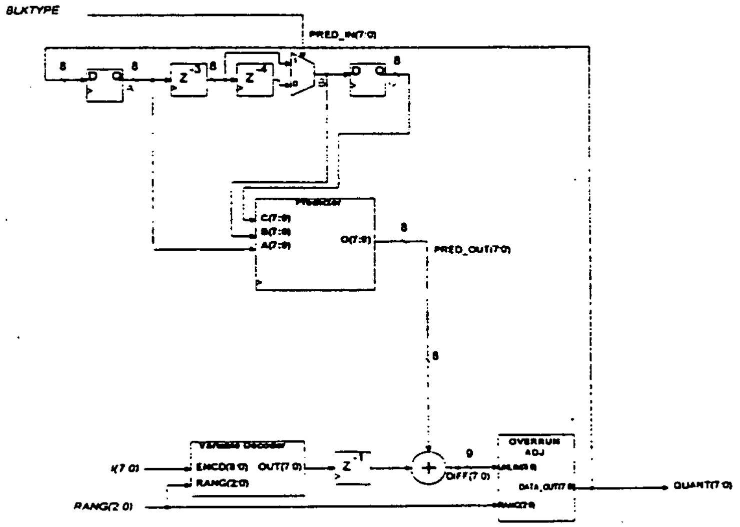


Figure G22- Decompression Function

G3.4.3. Inputs

CLK:
81 Mhz decompression clock

OUT_REQ:
This bit indicates the display section/MCU is reqesting compressed data.

BLKTYPE:
This control bit indicates the block type. '0' indicates an 8x8 luma block while '1' indicates a 4x4 chroma block.

D_INa(15:0):
First sixteen bit compressed data input in 8x8 luma field block format.

D_INb(15:0):
Second sixteen bit compressed data input in 8x8 luma field block format.

FIFO_ACKa:
This bit indicates compressed data is available from the first FIFO (a).

FIFO_ACKb:
This bit indicates compressed data is available from the second FIFO (b).

CLEAR:
This contol bit resets this section.

MODE:
This control bit indicates the current compression mode: 0 selects 2M/3 and 1 selects H/2M/2 mode.

G3.4.4. Outputs

D_OUT(7:0):
Decompressed eight bit data in 8x8/4x4 luma/chroma field block format.

FIFO_REQa:
This active high signal requests a new 16 bit work from FIFO *A*.

FIFO_REQb:
This active high signal requests a new 16 bit work from FIFO *B*.

OUT_ACK:
This signal indicates to the display section/MCU that the compressed data on *D_OUT(7:0)* is valid.

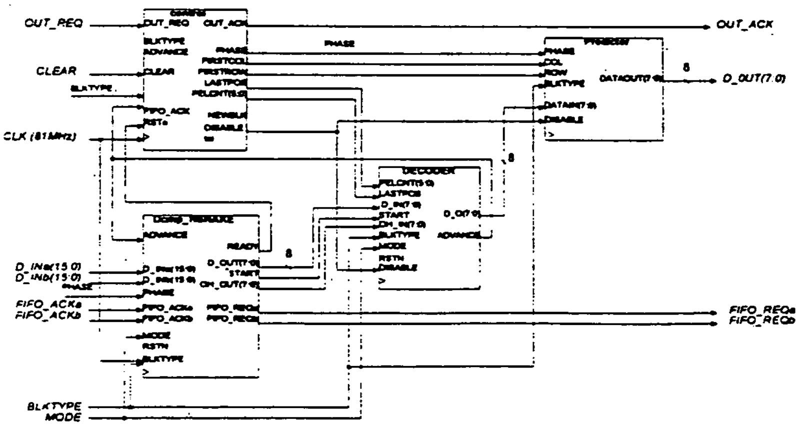


Figure G23- *Single Decompression Section*

G3.4.5. Decompress Predictor/DPCM

G3.4.5.1. Overview

The same adaptive prediction technique is used in the compression and decompression sections. The pixel to be coded is predicted by previously coded pixels (since they are known to the decompression section). The following figure shows the pixels used to make the prediction.

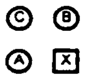


Figure G24- *Predictor Pixel Relationship*

In the figure X is the next pixel to be coded. A, B, and C are previously coded pixels known to the decompression circuit. A prediction of X is made using A, B, and C. The following pseudo code describes the algorithm which must be used:

```
if        $(|A-C|<e_1 \;\&\&\; |B-C| > e_2)$        $X_{pred} = B;$
else if   $(|B-C|<e_1 \;\&\&\; |A-C| > e_2)$        $X_{pred} = A;$
else                                                $X_{pred} = (A+B)/2;$
```

Note the above algorithm is only valid for pixels not in the first row or first column of the block. The exceptions are handled as follows:

- The first pixel of the block is not coded
- The pixels of the first row use A as the predictor
- The pixels of the first column use B as the predictor

G3.4.5.2. Inputs

DATAIN(7:0):

Eight bit quantized value of the current pixel.

COL:

This active high bit indicates the current pixel is located in the first column of a luma/chroma block.

ROW:

This active high bit indicates the current pixel is located in the first row of a luma/chroma block.

BLKTYPE:

This control bit indicates the block type. '0' indicates an 8x8 luma block while '1' indicates a 4x4 chroma block.

DISABLE:

This active high control bit causes the delay line to stop advancing data. The delay line does continue to alternate the phase of the current interleaved block.

G3.4.5.3. Outputs

DATAOUT(7:0):

Eight bit decompressed pixel value.

G3.4.5.4. R Bus Registers

G8(0) PredEnable:

This active high bit enables the predictor. When low, the predictor output is 0 and the registers are cleared to 0.

G3.4.5.5. Test Bus Registers

G3.4.5.5.1. Read Addresses:

G?(0) TestDataSelect:

This regsiter selects alternate signals to be connected to the *DATAOUT* bus. Note the DPCM loop continues to operate normally; only the output is affected.

00 - Normal operation
01 - *PRED_DLY()* is selected
02 - *DATAIN()* is selected

G3.4.5.5.2. Write Addresses:

G3.4.5.6. Description
This block incorporates the DPCM loop and the predictor section.

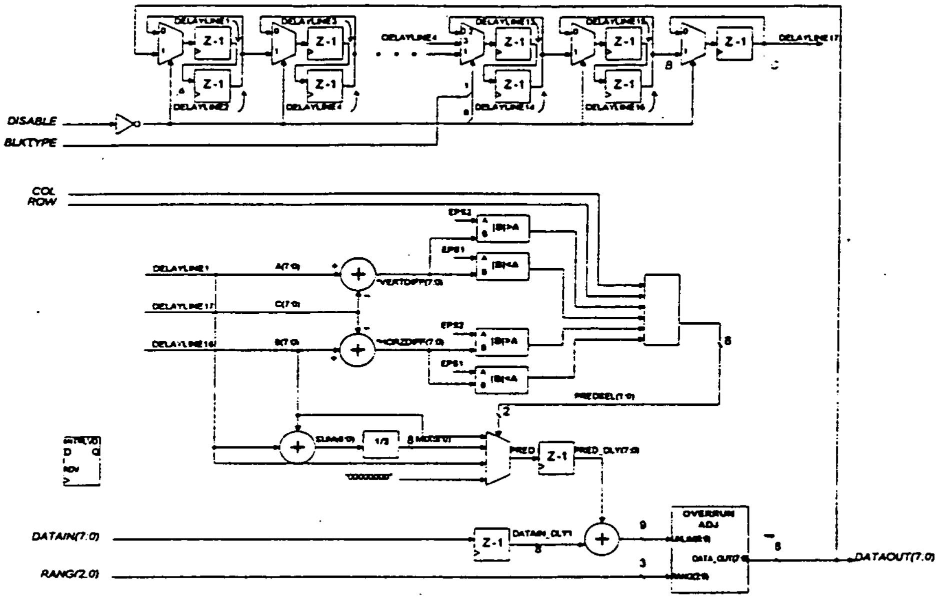


Figure G25- Predictor Block

3.4.6. Decompression Controller

3.4.6.1. Overview

G3.4.6.2. Inputs

FIFO_ACK:
This signal indicates the compressed data at the decoder input is valid. Should be called DATA_RDY

OUT_REQ:
This bit indicates the destination for the uncompressed data is ready for new data.

CLEAR:
This resets the state machine.

BLKTYPE:
This control bit indicates the block type. '0' indicates an 8x8 luma block while '1' indicates a 4x4 chroma block.

OH_ENA:
This active high bit indicates the data at the decompress VLD output is overhead data.

CLK:
81 Mhz decompression clock. This clock is always active.

G3.4.6.3. Outputs

IN_REQ:
This singal indicates the decoder block is ready for new data.

OUT_ACK:
This active high signal indicates the uncompressed data at the output is valid.

PELCNT(5:0):
This bus indicates the pixel number of the current data. The first pixel of a compressed block is 0. The last pixel for a luma block is 63 and for a chroma block is 15.

FIRSTROW:
This bit indicates to the predictor that current pixel is in the first row of the luma/chroma block For luma this occurs when PELCNT is in the range [0,7]. For chroma, PELCNT is in the range [0,3].

FIRSTCOL:
This bit indicates to the predictor that current pixel is in the first column of the luma/chroma. For luma this occurs when PELCNT is wither (0, 8, 16, 24, ..., or 56). For chroma, PELCNT is (0, 4, 8, or 12)..

LASTPOS:
For luma and chroma blocks, this bit indicates the current pixel is in the last row of pixels or is the last pixel of the last three rows. For luma, this occurs when PELCNT is in the range [56,63] or equal to 55 or 47. For chroma, PELCNT is in the range [12, 15] or equal to 11 or 7.

PHASE:
This bit indicates the current interleaved block that is active. PHASE alternates on every clock cycle that the decompressor is active. The block active after is reset is defined to be phase 0.

DISABLE:
This active high signal disables the decompress VLD and output pipeline. This should go active when the decompress pipe is full of data and *IN_REQ* is not active.

RSTN:
This active low bit resets this section.

CLK:
81 Mhz decompression clock. This clock is always active.

G3.4.6.4. Description

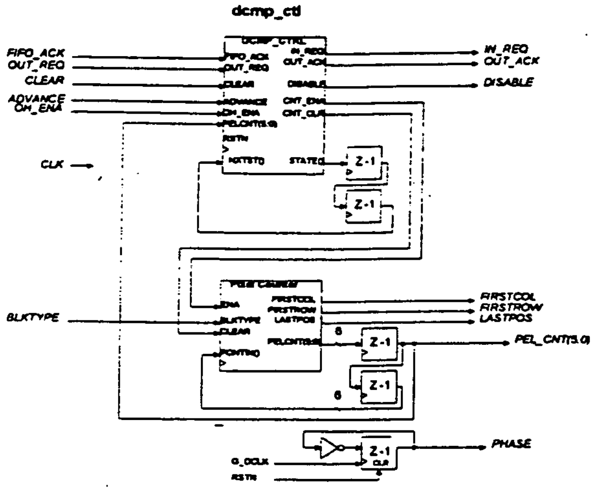


Figure G26- Decompression Controller Section

The controller provides the following functions:

- Tracks the position in an 8x8/4x4 block of the current pixel via a counter. The pixel counter is reset on RSTN and counts the number of pixels/cycles of the current block. This then generates the signals LASTPOS, FIRSTCOL, and FIRSTROW for the predictor and decoder blocks.

- Two sets of handshaking signals are generated, FIFO_REQ and FIFO_ACK with the handshake controller section, and OUT_REQ and OUT_ACK with the MCU or Display sections.
- Generate the signal *PHASE* which tracks which of the interleaved blocks to be decompressed is currently active.

- Detects when the barrel shifter should be emptied. If the barrel shifter generates an *ADVANCE* pulse, which indicates data has been clocked into the shifter, and *FIFO_ACK (DATA_RDY)* is low, random data has been loaded in the barrel shifter. The existing data must be decoded and the barrel shifter must then reload valid data.

G3.4.7.Decompression Handshake Controller

G3.4.7.1. Overview
3.4.7.2. Inputs

FIFO_ACKa:
This active high signal indicates the data on D_INa is valid.

FIFO_ACKb:
This active high signal indicates the data on D_INb is valid.

ADVANCE:
This bit indicates the barrel shifter has latched in the data on *D_OUT*.

PHASE:
This bit indicates the current interleaved block that is active. PHASE alternates on every clock cycle that the decompressor is active. The block active after is reset is defined to be phase 0.

D_INa(15:0):
First sixteen bit compressed data input in field macroblock format.

D_INb(15:0):
First sixteen bit compressed data input in field macroblock format.

BLKTYPE:
This control bit indicates the block type. '0' indicates an 8x8 luma block while '1' indicates a 4x4 chroma block.

MODE:
This control bit indicates the current compression mode: 0 selects 2M/3 and 1 selects H/2M/2 mode.

RSTN:
This active low bit resets this section.

CLK:
81 Mhz decompression clock. This clock is always active.

G3.4.7.3. Outputs

FIFO_REQa:
This active high signal requests a 16 bit data work from FIFO *A*.

FIFO_REQb:
This active high signal requests a 16 bit data work from FIFO *B*.

READY:
This active high signal indicates the data on D_OUT and OH_OUT (during overhead cycle) is valid.

D_OUT(7:0):
The eight bit compressed data on this bus alternates between the two data inputs. It also alternates between the most significant and least significant byte of each input on subsequent reads.

OH_OUT(7:0):
The eight bit compressed data on this bus alternates between the two data inputs. It contains the first eight bits of a compressed block and is only valid during the overhead cycle.

START:
This active high signal indicates that data on *D_OUT(7:0)* and *OH_OUT(7:0)* represents the first 16 bits of a new block of compressed data.

G3.4.7.4. Description

This block is the interface between the decompression controller and the two FIFOs. It performs several functions:

- It converts the *READY/ACK* handshaking with the decompression controller to the *REQ/ACK* handshaking with the FIFOS.
- It interleaves compressed data from two independent blocks by multiplexing between the two FIFO data inputs, *D_INa* and *D_INb*.
- It feeds the 16 bit data from the FIFOs eight bits at a time to the barrel shifter
- It tracks when a word read out of a given FIFO corresponds to the first 16 bits of a compressed block of data and, therefore, contains overhead data.
- It provides the first 16 bits of a compressed block to the barrel shifter at the same time. All other data is provided 8 bits at a time. This reduces the cycles lost decoding overhead data to one cycle per luma/chroma block.

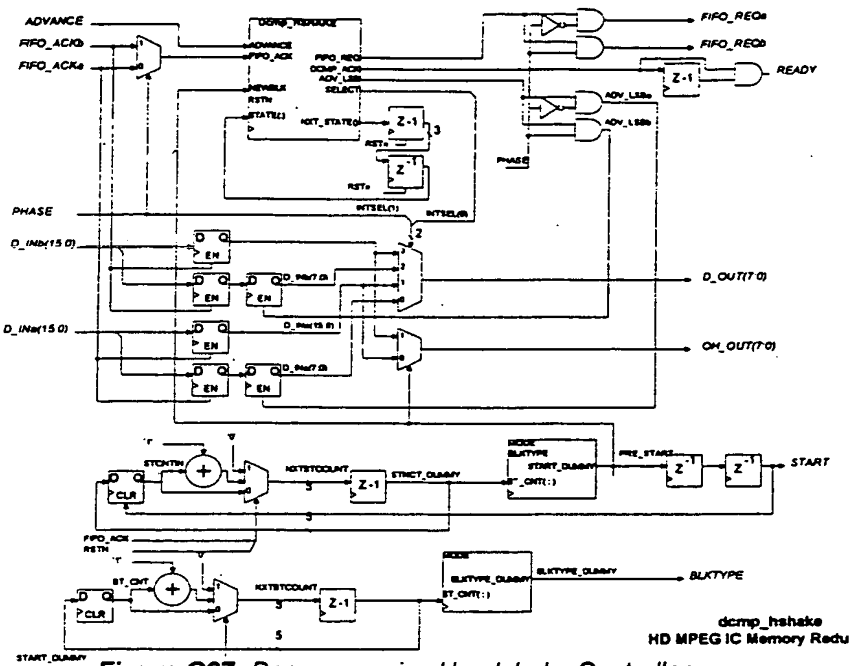


Figure G27- Decompression Handshake Controller

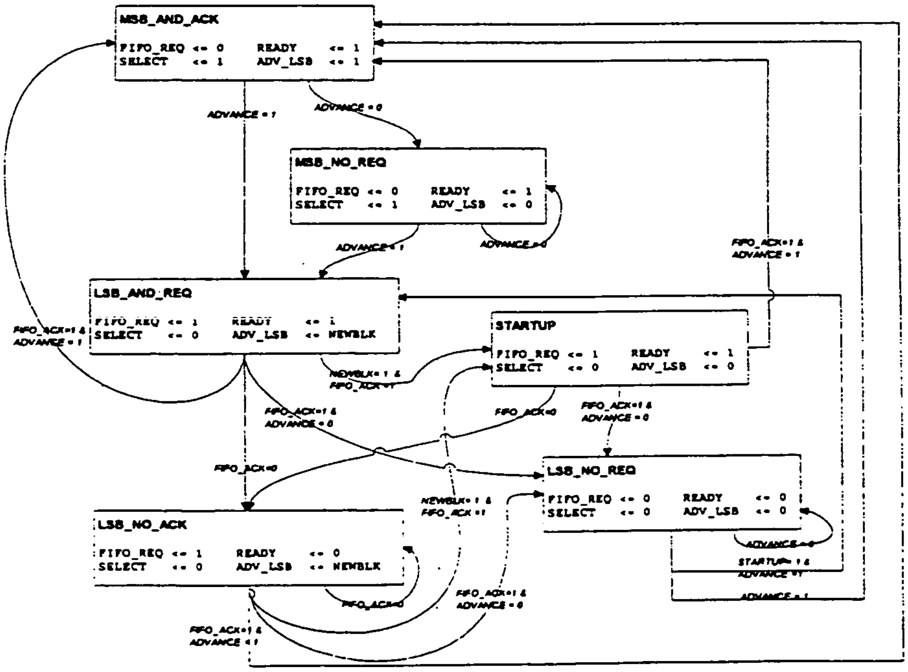


Figure G28- State Machine for Handshake Controller

G3.4.8. Decoder

G3.4.8.1. Overview

G3.4.8.2. Inputs

D_IN(7:0):
This eight bit bus provides the compressed data to the barrel shifter.

OH_IN(7:0):
When START is high, this eight bit bus provides the first eight bits of data from a compressed block. This data always contains overhead information.

BLKTYPE:
This bit indicates the block type. '0' indicates an 8x8 luma block while '1' indicates a 4x4 chroma block.

DISALBE:
This active high bit disables the barrel shifter. This should go active when the decompress pipe is full of data and no data is requested by the decompressed data's desitnaiont section.

START:
This active high signal indicates that data on *D_IN(7:0)* and *OH_IN(7:0)* represents the first 16 bits of a new block of compressed data.

MODE:
This control bit indicates the current compression mode: 0 selects 2M/3 and 1 selects H/2M/2 mode.

LASTPOS:
This control bit indicates that if bit savings is not positive for the current block, this pixel should be quantized with the 3 bit quantizer.

PEL_COUNT(5:0):
This eight bit bus provides the position in a compressed block of the current pixel. This is used to determine which quantization table was used. (NOT YET IN VHDL)

RSTN:
This active low bit resets this section.

CLK:
81 Mhz decompression clock. This clock is always active.

G3.4.8.3. Outputs

DECODE_OUT(7:0):
Eight bit decoded data in field luma/chroma block format. This data is passed to the decompression DPCM loop.

RANG(2:0):
This bus indicates the range of the given quantizer used on the block currently being decoded. This is obtained from the overhead bits of the block.

OH_ENA:
This active high bit indicates the data on *DECODE_OUT* is overhead data from a new block of data.

ADVANCE::
This active high bit indicates the barrel shifter has clocked in the data present on *D_IN*.

G3.4.8.4. Description

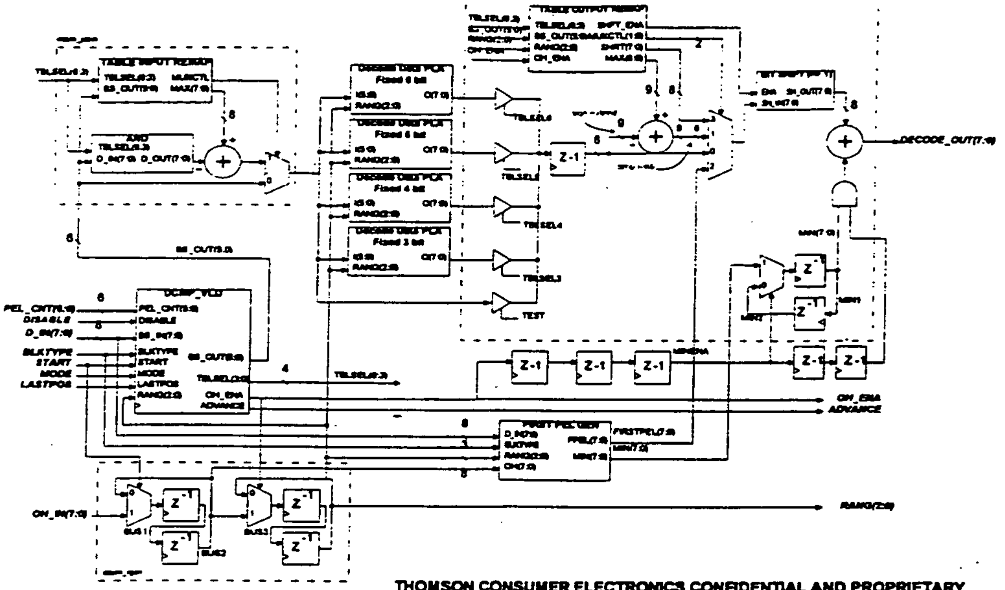


Figure G29- Decoder Block

For a description of decompression operation, the first clock of decompression is called the overhead cycle. The second clock is data1, the third data2, etc..

START goes high when the first 16 bits of data of a new block are present. The first 16 bits are read in together to minimize the number of cycles spent on overhead bits. As table G1 shows, by always stripping off the first eight bits of a compressed block, the remaining bits can be shifted out in one clock cycle.

During the overhead cycle, the range and minimum need to be setup and the overhead bits need to be shifted through the barrel shifter.

A variable number of overhead bits are included with every luma/chroma block to provide information on how the block was compressed. Table G1 specifies the format of the overhead bits and is repeated below.

The first eight bits of a compressed data block is immediately stripped off. The remaining 0 to 6 overhead bits can then be handled by the 6 bit wide barrel shifter in the decoder.

There are 4 decoder tables included in this section each of which contains 3 - 4 ranges:

Decompress 3 bit Table, 4 ranges (50% Mode)
Decompress 4 bit Table, 4 ranges (50% Mode)
Decompress 5 bit Table, 3 ranges (25% and 50% Mode)
Decompress 6 bit Table, 1 ranges (25% Mode)

		Input bits	Output Bits	Symmetric
Quantization	dmpr_qtbl3	2+3=5	8	Yes
	dmpr_qtbl4	2+4=6	8	Yes
	dmpr_qtbl5	2+5=7	8	Yes
	dmpr_qtbl6	2+6=8	8	Yes

The tables convert the encoded index into the quantized value generated by the compressors DPCM loop.

Table G5- Decoding Tables

3 Bit Table

Index	Level #
10	3, 5
01	2, 6
00	1, 7
* 11	0, 4

4 Bit Table

Index	Level #
110	6, 8
101	5, 9
100	4, 10
011	3, 11
010	2, 12
000	1
001	13
* 111	0, 7

5 Bit Table

Index	Level #
1110	12, 14
1101	11, 15
1100	10, 16
1011	9, 17
1010	8, 18
1001	7, 19
1000	6, 20
0111	5, 21
0110	4, 22
0101	3, 23
0100	2, 24
0000	1
0011	25
* 1111	0, 13

* Level not included in table. Output Remapper generates these levels.

The quantize-to-encode and encode-to-quantize tables are symmetric; therefore, only half of the tables are needed in the IC. An output remapping block generates the top half of the table from the lower half. The lsb of the input is used to separate the top and bottom half of the table.

The Table Input Remap performs the following:

Table G6- Table Output Remapping

Inputs	Outputs	
RANG(2:0)	MAX(7:0)	SHFT_ENA
000	16	0
010	64	1
011	64	0
100	192	1
101	192	0
110	256	1
111	256	0

***Table G7-** Table Output Remapping, con't*

Inputs			Outputs	
OVHD_ENA	**TBLSEL(6:3)**	**BS_OUT()**	**SHORT(7:0)**	**MUXCTL(1:0)**
0	0000	X	X	00
0	0001	110	0	11
0	"	111	MAX(7:0)/2	11
0	"	else	X	0 BS_OUT(0)
0	0010	1110	0	11
0	"	1111	MAX(7:0)/2	11
0	"	else	X	0 BS_OUT(0)
0	0100	11110	0	11
0	"	11111	MAX(7:0)/2	11
0	"	else	X	0 BS_OUT(0)
0	1000	000000	0	11
0	"	111111	MAX(7:0)/2	11
0	"	else	X	0 BS_OUT(0)
1	X	X	X	10

The controller is responsible for the following:
Selecting correct table base on:
25% or 50% based on control bus and block type

In the 50% mode, the Length controller uses the following algorithm:
A *START* resets the *BIT_SAVINGS* counter and enables the fixed length PLA output. The initial shift value depends on the Compression Mode, Block Type, and the overhead bit values as shown in Table G1

G3.4.9. Barrel Shifter

G3.4.9.1. Overview
G3.4.9.2. Inputs

I(7:0):
This eight bit bus provides the compressed data to the barrel shifter.

RANG(2:0):
This bus indicates the range of the given quantizer used on the block currently being decoded.

BLKTYPE:
This bit indicates the block type. '0' indicates an 8x8 luma block while '1' indicates a 4x4 chroma block.

DISALBE:
This active high bit disables the barrel shifter. The shift value for the barrel shifter is set to 0 when this bit is active.

MODE:
This control bit indicates the current compression mode: 0 selects 2M/3 and 1 selects H/2M/2 mode.

START:
This active high signal indicates that data on *I(7:0)* represents the second 8 bits of a new block of compressed data. This may or may not correspond to overhead data depending on *BLKTYPE* and *MODE.*

LASTPOS:
This control bit indicates that if bit savings is not positive for the current block, this pixel should be quantized with the 3 bit quantizer.

PEL_COUNT(5:0):
This eight bit bus provides the position in a compressed block of the current pixel. This is used to determine which quantization table was used. (NOT YET IN VHDL)

RSTN:
This active low bit resets this section.

CLK:
81 Mhz decompression clock. This clock is always active.

TST_ADV:
This active high test bit forces the barrel shifter to advance 8 bits.

G3.4.9.3. Outputs

BS_OUT(7:0):
Eight bit decoded data in field luma/chroma block format. This data is passed to the decompression DPCM loop.

MINENA:
This actibr high bit updates the minimun pixel value stored in the *dcpm_plaout* block. This is timed to occur when a new block is available at the output.

OVERHEAD_ENA:
This active high bit indicates the data on *DECODE_OUT* is overhead data from a new block of data.

ADVANCE::
This active high bit indicates the barrel shifter has clocked in the data present on *D_IN*.

TBL_SEL(3:0):
These four active high control bits enable the various decoding tables: 3 bit, 4 bit, 5 bit, or 6 bit. The usage is data dependent.

G3.4.9.4. Description

There are three eight bits registers that supply the 19 bits of data to the barrel shifter block. When the first byte of a new block is shifted into register *BS_IN3D* and all input data to the barrel shifter is from the new block, *OH_ENA* will be high for one clock cycle. Since every compressed block will be a multiple of 8 bits (32 bytes or 48 bytes), *OVERHEAD_ENA* will go high on cycle Data1 of a new block. For example, if the last compressed datum is 4 bits wide, the LUT will produce a length of 4, and the barrel shifter will shift in an additional 4 bits and since this will is on a byte boundary the flag will be set to advance the BS_IN registers on the next cycle.

Special consideration needs to be given to the first block of data after a clear. Since there is invalid data provided to the barrel shifter, the output of the LUT is undefined. Therefore a reset forces the SM into a initial state that shifts 8 bits per cycle until valid data is present.

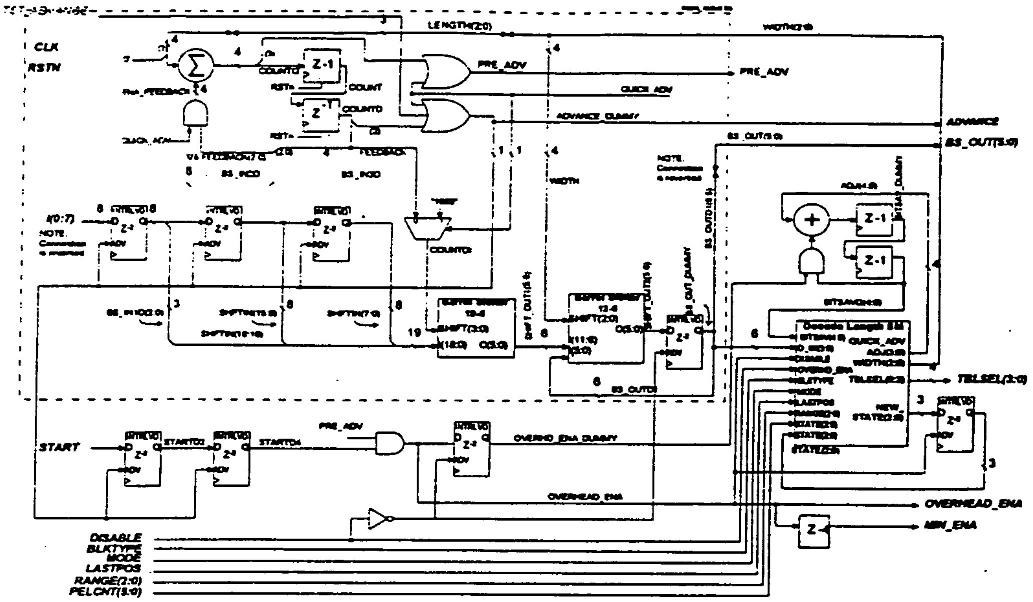


Figure G30- Decoder Barrel Shifter (dcmp_brlshft.vhd)

Table G8- Decode State Machine Outputs during Header

	MODE=0			MODE=1		
RANGE	ADJ()	WIDTH(2:0)	OVERHEAD()	ADJ()	WIDTH(2:0)	OVERHEAD( )
0	-10	6	10	0	5	10
2	-6	2	6	0	5	6
3	-7	3	7	0	5	7
4	-8	4	8	0	5	8
5	-9	5	8	0	5	8
6	-8	4	9	0	5	9
7	-6	2	6	0	5	6

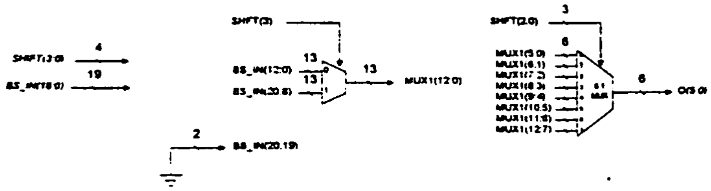


Figure G31- 19-6 Barrel Shifter

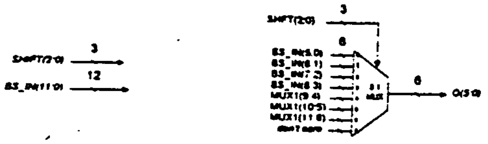


Figure G32- 12 - 6 Barrel Shifter Block

G4. TIMING

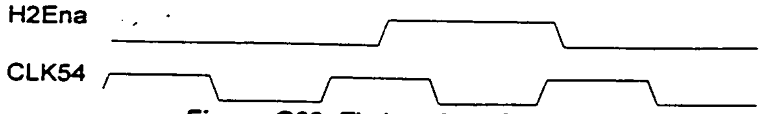


Figure G33- Timing of H2 Enable

HD MPEG VIDEO DECODER

APPENDIX H

D1 INTERFACE

Thomson Consumer Electronics, Inc.
Indianapolis, Indiana, USA

Revision No. 2.3

THESE DRAWINGS AND SPECIFICATIONS ARE THE PROPERTY OF THOMSON CONSUMER ELECTRONICS, INC. AND SHALL NOT BE REPRODUCED OR COPIED OR USED AS THE BASIS FOR THE MANUFACTURE OR SALE OF APPARATUS OR DEVICES WITHOUT WRITTEN PERMISSION.

TCE PROPRIETARY AND CONFIDENTIAL

TABLE OF CONTENTS

FIGURES

H1. STANDARD DEFINITION PIXEL INTERFACE

H1.1. Overview

The HD-MPEG IC will be used in television receiver applications which also support standard definition television from an analog channel. In order to provide a common and consistent display path under all conditions the HD-MPEG IC must provide a means for writing digitized component video from an external source into the local memory. The data will be written in a format consistent with the output of the MPEG decoding process.

The interface will be via a 9 pin port conforming to the parallel D1 interface standard. The 9 pins include 8 data pins carrying multiplexed YC_RC_B pixels in 4:2:2 format and a 27 MHz clock. A conversion from 4:2:2 format to 4:2:0 format is done by the HD-MPEG IC. Two additional pins are needed to handle video data that does not have sync information imbedded per CCIR656.

All the sync information, user data, pan and scan information, and video data are combined in a single 8 bit data stream. The D1 Interface extracts this information and uses it to identify the lines and pixels to be written to memory. The location within the memory is programmed by the external micro controller.

The YC_RC_B pixels received through the D1 Interface are in the 4:2:2 format. Conversion to 4:2:0 format is done by simply chroma line decimation. Odd numbered lines of chroma (first active line of each field is line 0) are dropped (not written to memory) from each field.

H1.2. Inputs

V_DATA (7:0) *(* D1IN*(7:0)):*
D1 4:2:2 format data. This data input may be either CCIR656 format data or the 27MHz NTSC multiplexed YC_RC_B associated with V_HSYNC and V_VSYNC below.

*V_HSYNC (*D1_H*):*
Horizontal signal for NTSC data. This bit is low during blanking and high during active video. This will be used if the D1 Interface must accept 27 MHz NTSC multiplexed YC_RC_B that does not comply with CCIR656.

*V_VSYNC (*D1_V*):*
Vertical signal for NTSC data. This will be used if the D1 INTERFACE must accept 27 MHz NTSC multiplexed YC_RC_B that does not comply with CCIR656.

*V_CLKn (*D1_EN_CLK*):*
27 MHz clock externally generated. This clock is gated and enabled by the D1 enable control bit.

V_RST_STRETCH:
Vertical sync signal from the display section. This pulse is double buffered to create V_RST_D1_CLK in the D1 clock domain.

H1.3. Outputs

YC_DATA(7:0):
This eight bit bus is a one clock delayed version of D1IN(7:0). This bus is connected to both the luma and chroma FIFOs.

HSyncOut:
This single 27 MHz clock wide active high signal indicates the start of a new active line of video to be written to the luma and chroma FIFOs.

VSyncOut:
This single 27 MHz clock wide active high signal indicates the start of a new field of data to be written to the luma and chroma FIFOs. This signal always precedes HSyncout.

UpperField:
This signal indicates whether the data being written into the FIFOs is from the upper or lower field. A high indicates upper field, low indicates lower field.

CDataValid:
This active high signal indicates that data on YC_DATA is valid chroma data.

YDataValid:
This active high signal indicates that data on YC_DATA is valid luma data.

H1.4. Bus Registers

Host Bus Addresses:

A0(0) D1 Enable:
This active high bit enables the D1 interface. When low, the input D1CLK is gated, and the D1 circuitry is reset.

A0(1) MODE
This bit controls the data format expected. '0' indicates a standard CCIR-656 bitsream is present on the eight D1IN pins and the D1CLK pin. '1' indicates that horizontal and vertical information is not embedded in the stream.

A0(2) PAL
This bit also controls the data format expected. '0' indicates an NTSC-525 line input bitstream is present on the eight D1IN pins. '1' indicates a PAL-625 input bitstream is present on the eight D1IN pins.

A0(3) V_EDGE
This bit determines which edge of the V_VSYNC signal is to be used to resolve the LineCount and Upperfield output values. '0' indicates rising edge, '1' indicates falling edge.

A1(7:0) LineCount:
This eight bit port corresponds to the upper eight bits of the line counter for the D1 Video bit stream. It is updated on the rising edge of V_RST_D1_CLK.

A2(1:0) D1L1(9:8):
The two MSBs of the lower horizontal count limit. When V_VSYNC arrives, the actual horizontal count is compared with this limit to determine field type.

A3(7:0) D1L1(7:0):
The eight LSBs of the lower horizontal count limit. When V_VSYNC arrives, the actual horizontal count is compared with this limit to determine field type.

A4(1:0) D1L2(10:8):
The three MSBs of the upper horizontal count limit. When V_VSYNC arrives, the actual horizontal count is compared with this limit to determine field type.

A5(7:0) D1L2(7:0):
The eight LSBs of the upper horizontal count limit. When V_VSYNC arrives, the actual horizontal count is compared with this limit to determine field type.

Test Bus Read Addresses:

1C2(0) D1H:
This read bit is the Horizontal signal from the D1 decoder.
1C2(1) D1V:
This read bit is the Vertical signal from the D1 decoder.
1C2(2) D1F:
This read bit is the Field type signal from the D1 decoder.
1C1(7:0) TestData:
This 8-bit bus is valid YC_DATA output from the the D1 decoder.

Test Bus Write Addresses:

1C0(1:0) TestMode:
This test register selects various bypass modes:
00: Normal Operation
01: All data from D1 port is placed in Luma FIFO
02: All data from D1 port is placed in Chroma FIFO
03: All data from D1 port is placed in both FIFOs

Debug Bus Addresses:

20(10:0) HS_CNT(10:0):
The horizontal sample counter value.
20(20:11) FIELD_LCNT(8:0):
The field line counter value.
20(21) FLCNT_START:
Signal that resets and enables the field line counter.
20(22) D1_EN_CLK:
The 27 MHz clock.

21(1:0) LCNT2_LSBS(1:0):
The two LSBs of the linecount for the external sync circuitry.
21(9:2) LINECOUNT2(7:0):
The upper eight bits of the linecount for the external sync circuitry.
21(10) LCNT_TWO:
This bit loads the line counter based upon whether or not V_VSYNC in arrived before or after the corresponding H_SYNC.
21(11) LCNT_START:
This bit resets and starts the line counter for the external sync circuitry.
21(12) D1_EN_CLK:
The 27 MHz clock.

22(1:0) LCNT1_LSBS(1:0):
The two LSBs of the linecount for the CCIR656 circuitry.
22(9:2) LINECOUNT1(7:0):
The upper eight bits of the linecount for the CCIR656 circuitry.
22(10) START_LINE:
This bit resets and starts the line counter for the CCIR656 circuitry.
22(12) D1_EN_CLK:
The 27 MHz clock.

H1.5. Description

The D1 interface consists of a state machine and a line counter that tracks the incoming video bit stream. This block has two modes of operation. In the first mode, a CCIR-656 compliant bit stream is present at the 8 data pins and the *D1CLK* pin. Horizontal and vertical timing are embedded in the bit stream, as well as an indication for upper and lower fields.

In the second mode, blanking information is provided via the *D1_V* and *D1_H* pins. The *D1_H* pin is low during the horizontal blanking interval and high during active video. The *D1_V* pin will provide vertical synchronization. Field type must be determined based on the relative timing of the *D1_V* and *D1_H* signals. For an NTSC system, the vertical sync, *D1_V*, will occur at approximately the same time as the *D1_H* signal for the upper field, and will occur during the middle of a line (approximately centered between consecutive *D1_H* signals) for the lower field. For a PAL system the relationships of *D1_V* and *D1_H* for upper and lower fields are the opposite of those for NTSC. For each case, a horizontal sample counter is reset and started upon the arrival of each *D1_H*. When *D1_V* arrives, the value of the counter is compared against a pair of limits to determine where it arrived relative to *D1_H*, and the field type is determined.

The D1 data is registered and passed on to the luma and chroma FIFOs via the bus *YC_DATA(7:0)*. The enable signals *YDATAVALID* and *CDATAVALID* indicate what data is present on this bus. The order of the multiplexed video data of a video line is C_B Y C_R Y C_B Y C_R Y ....

The D1 interface also supplies synchronization information to the control micro. The rising edge of *DSPL_VERTICAL* latches the current value of the line counter. This value is readable via the host bus. This information allows the micro to calculate the relative phase of the incoming bitstream and the vertical section.

This section is enabled by the host bus control bit D1 Enable. When this bit is low, the input clock, *V_CLKn* is gated to disable this section. The *D1_Enable* signal in the low state generates a reset for the D1 section and *D1_Enable* in the high state enables the *V_CLKn*.

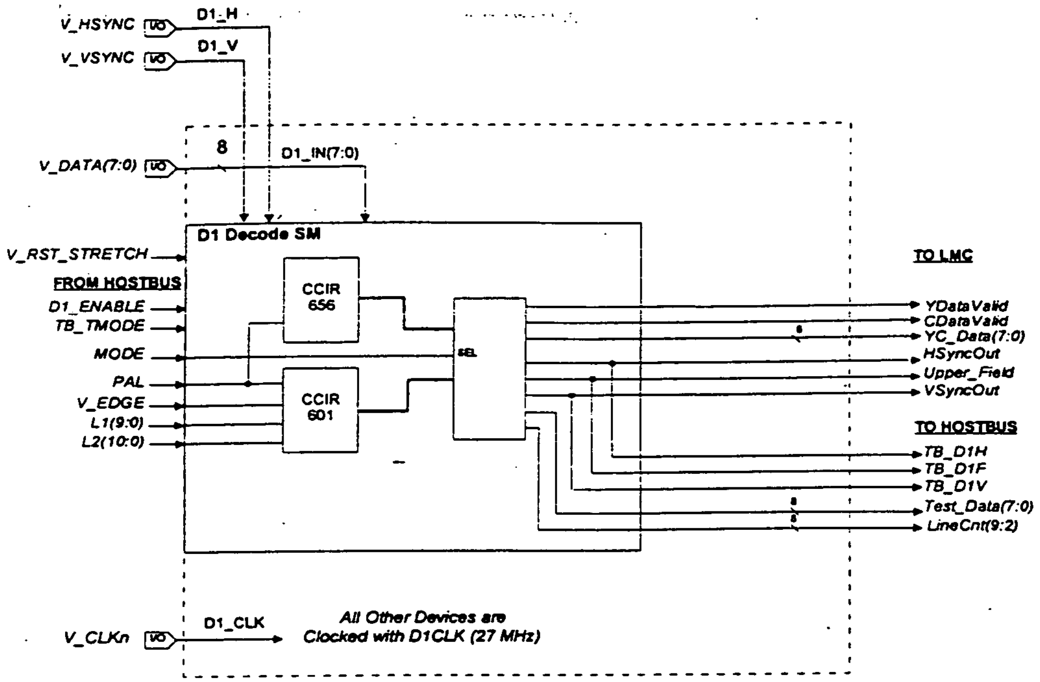


Figure H1- D1 Interface Block Diagram

H1.6. Interface Timing

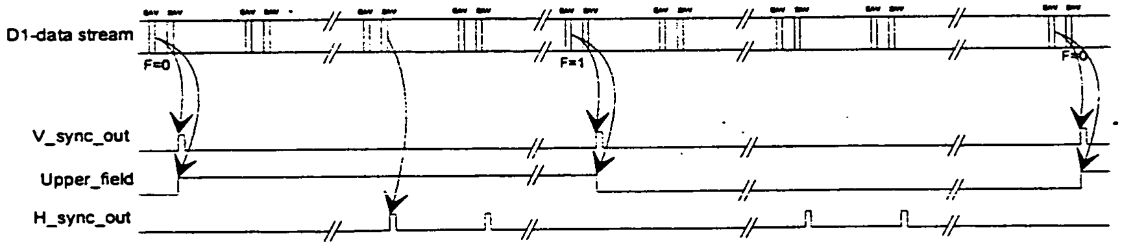


Figure H2- CCIR656 Sync Extraction

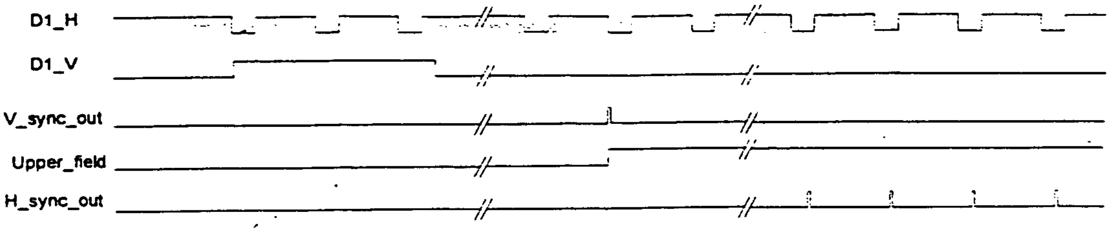


Figure H3- H and V Timing

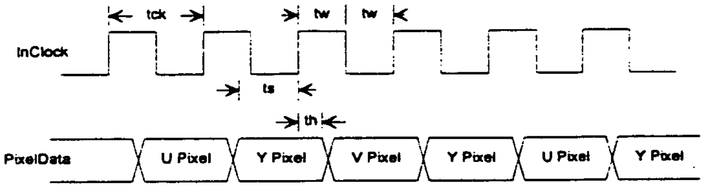


Figure H4 - Input Data Timing

parameter	min	units
tck	37	ns
tw	12	ns
ts	3	ns
th	2	ns

HD MPEG VIDEO DECODER

APPENDIX I

VARIABLE LENGTH DECODER

Thomson Consumer Electronics, Inc.
Indianapolis, Indiana, USA

Revision No. 2.1

THESE DRAWINGS AND SPECIFICATIONS ARE THE PROPERTY OF THOMSON CONSUMER ELECTRONICS, INC. AND SHALL NOT BE REPRODUCED OR COPIED OR USED AS THE BASIS FOR THE MANUFACTURE OR SALE OF APPARATUS OR DEVICES WITHOUT WRITTEN PERMISSION.

TCE PROPRIETARY AND CONFIDENTIAL

TABLE OF CONTENTS

I1. OVERVIEW

The VLD is the first stage of the decoding pipeline. It receives data from the bit buffer , decodes the variable length codes and delivers information to the two RLD/IQ/IDCT pipelines. The VLD performs the following functions:

- Synchronize the MPEG video bit stream after the bit buffer is flushed
- Variable length decode the slice layer, macroblock layer and block layer
- Decode the MBA and process skipped macroblocks
- Decode the motion vectors and deliver them to the motion compensation unit
- Detects bit stream errors and conceals corrupted and missing data.

The interfaces to the VLD block are shown below(figure I1). The VLD uses a 54 Mhz clock(decodes minimum 99.7 M codeword per second). The clock input is gated with control signals from the input FIFO (data available), and pipe control signals (pipeline stop, etc.). There is a Sequence reset (reset to next Sequence start code) , a Picture reset (reset to next picture start code), Dsync (start decode synchronization), and a skip picture command input.

The compressed data is input from the LMC through a FIFO that operates on 128 bit words. Also, the VLD receives decoding information from the upper layers of the bit stream (Sequence, GOP, Picture layers). This data is parsed by the external micro controller and loaded into registers in the IC.

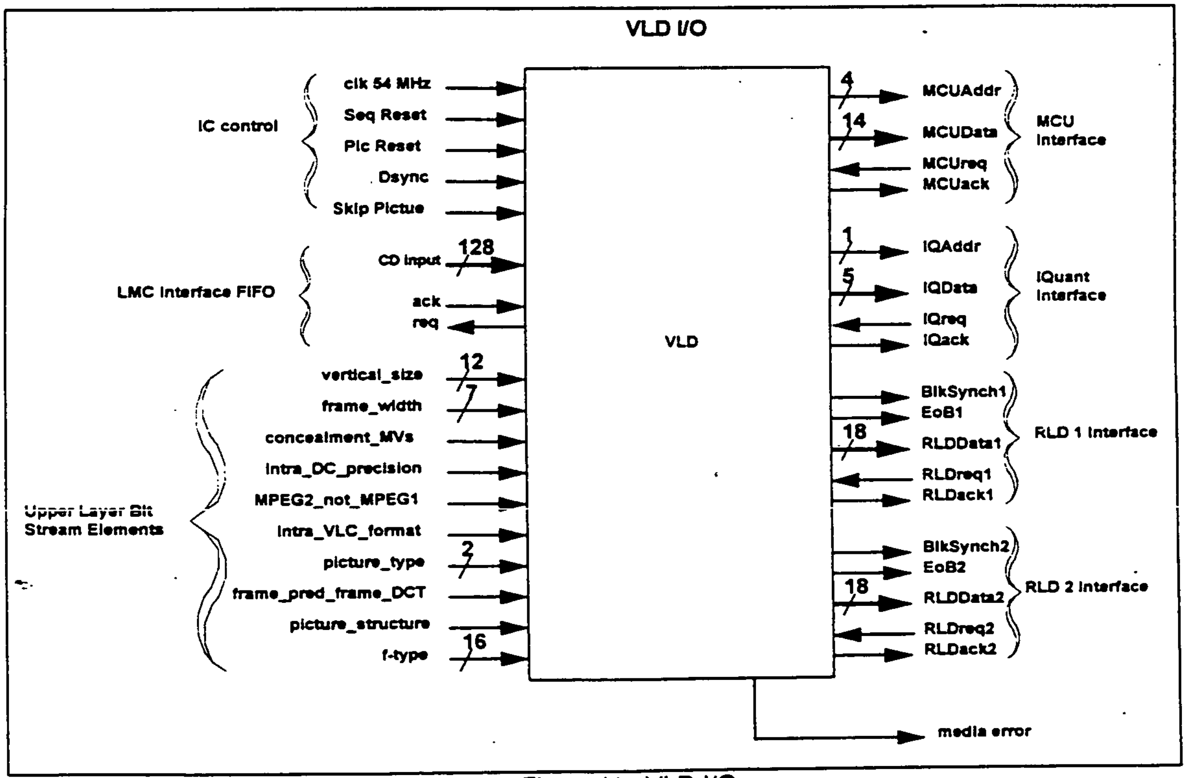


Figure I1 : VLD I/O

There are four output interfaces from the VLD. First, there are two pipeline interfaces (RLD1 and RLD2). DC and AC DCT coefficient data are transmitted on this bus to the run length decoders (RLD). These interfaces are composed of a

20 bit data bus (12 bit value, 6 bit run), a block synchonization(Bsynch), and end of block signal (EOB). Second, there is an interface to the MCU which is composed of a 4 bit address and a 14 bit data bus. The parsed and decoded motion vectors from the VLD are transmitted to the MCU on this bus. Finally, there is the Iquant bus which is for transmitting quant values from the VLD to the Inverse Quantizer.

An error signal is used to signal the pipeline when the VLD detects an error in the bitstream (media error code, or wrong MPEG data). The VLD runs in error concealment mode until the next MPEG synchronization point.

The top level block diagram of the VLD is shown below(figure I2). Data is processed by the Barrel Shift Circuit and Length LUT's in order to determine where the next codeword boundary is, and to keep enough data presented to the Value LUT. The Value LUT is a pipelined table that can decode up to two codewords per clock cycle. This output is then serialized into a Ping-Pong buffer structure. Every odd macroblock is fed to the Pipe 1 stage, and every even macroblock is sent to the Pipe 2 stage. A macroblock type (MBT) and motion vector (MV) delay line receives data for both odd and even macroblocks.. It is used for error concealment by copying the motion vectors from top adjacent macroblocks to the missing macroblock data. The state machine is made up of several sub state machines. One to control the decoding of codewords, one to process the macroblock address calculations, and one to decode the motion vector data.

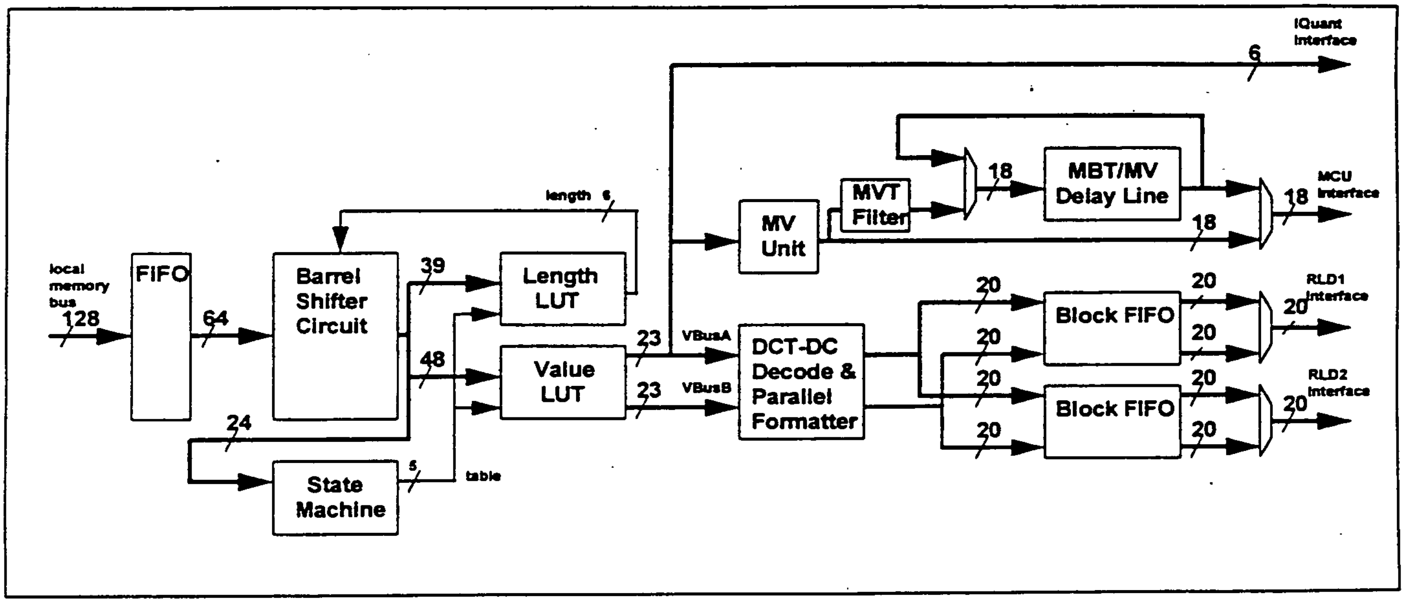


Figure I2 VLD Top Level Diagram

I2.PERFORMANCE

The VLD must meet two performance requirements. First, it must be capable of decoding a worst case frame in a frame period. For the Grand Alliance application this requires a decode rate of 97.9 Mega codewords per second. This assumes a 1920x1088 resolution with 1 slice per macroblock, 4 motion vectors per macroblock, and 64 coefficients per block. The calculation is

[(1920x1088x1.5) + (1920x1088/256)x(2+14)] x 30 = 97.9 Mega codewords per second.

Second, the VLD must not cause external memory accesses to be waited beyond their design margins. This means that the VLD should attempt to decode at a nominal linear rate or faster.

In order to achieve these performance requirements, the VLD was designed with the following constraints when decoding AC coefficients:

1) escape codes are decoded in a single clock cycle.
2) a minimum of 2 AC coefficients are decoded per clock cycle.

Constraint 1) is achieved by designing the barrel shift path and LUT's to decode the escape code, run length and value length (24 total bits) in one clock cycle. This requires a minimum 24 bit path through the barrel shifters and a wider LUT.

Constraint 2) is achieved by decoding pairs of AC coefficients that have runs of 0,0 or 0,1 in one clock cycle. (this includes escape codes with runs of 0 and 1). In other words, AC coefficient codewords with run lengths greater than 1 are decoded in one cycle, and consecutive 0,0 run length AC coefficients and consecutive 0,1 run length coefficients are decoded in one cycle. This guarantees that the average coefficient per cycle rate is 2 or greater (a 0,0 combination decoded in one cycle achieves a rate of 2 coefficient/cycle. A 0,1 combination achieves a rate of 3 coefficients per cycle. If the 0,1 combinations were not decoded together, then the worst case rate would only be 1.5 coefficients per cycle.)

With these design constraints and a clock rate of 54 MHz, the VLD can meet the two performance requirements. For a worst case frame with all 0 run coefficients the VLD will decode 2 coefficients per cycle. At this rate it is capable of decoding a maximum of 99.7 Mega codewords per second. This assumes a frame structure with 1 slice per macroblock, and 4 motion vectors per macroblock. The calculation is

[54 Mcycles/sec]x[384 coef / (192 + 14 + 2) cycles] = 99.7 Mega codewords per second.

This is a 2% margin for the worst case frame and the first performance requirement.

The second performance requirement is also met. The worst case frame size is 8,000,000 bits (constrained by the bit buffer size). Therefore the slowest VLD decode rate would be when it is decoding the largest codewords with the smallest runs. This corresponds to decoding all escape codes with a run of 1. Therefore the minimum number of one run codewords is 8,000,000/24 = 333,333 codewords. 666,666 coefficients were produced by this decoding in 361109 cycles (333,333 + 1736*[14+2]). Therefore 666,666/(1920x1088x1.5) = 21.3% of the coefficients were decoded in 361109/1,800,000 = 20.0% of total alloted time for decode. This shows that the VLD is always ahead of the ideal linear decode rate and therefore has 1.3% margin on the second performance requirement.

IMPORTANT NOTE: The performance requirements of the VLD are optimized for MPEG2 bit streams. MPEG1 bit streams are limited to a frame size of 720x576. Therefore, the VLD decodes MPEG1 espape codes in two cycles. This is done to save area in the barrel shift circuit(since MPEG1 escape codes are 8 bits larger than MPEG2 escape codes).

13. BARREL SHIFT CIRCUIT

The barrel shift circuit in figure 13. It is the most time critical part of the VLD. The role of this circuit is to present enough data to the LUT's in order to decode the next codeword or codewords. This circuit is designed so that under the worst case codeword combination, there is never an extra clock needed. That is, this circuit will decode a minimum of one codeword per clock cycle. (NOTE: for MPEG1 bit streams, the escape codes are decoded in two clocks. This can be done since the MPEG1 bit streams are limited to a frame size of 720x576 or less).

The input stage of the barrel shift circuit consists of the local memory bus FIFO, and two 32 bit registers. The local memory bus feeds data to the FIFO from the bit buffer stored in external DRAM. The local memory controller must supply compressed data to this input FIFO so that the VLD never needs to stop. Stopping the VLD for a long enough period can cause the VLD to not complete its decoding in a frame time. Also, stopping the VLD can cause the pipelines to lose time also.

Since the local memory bus is 128 bit wide, the FIFO is organized as 128 bit input, and 64 bit output. It is not yet determined if the actual FIFO will be 128 bit output. Therefore, two 64 bit registers for muxing of the 128 bit data to 64 bit width would be necessary. The 64 bit words are fed to barrel shifter 1 as data flows through the VLD. Data is requested based on a accumulator circuit that keeps track of the number of bits shifted. When the number of bits shifted exceeds 64, then the next 64 bit word is read in from the FIFO circuit.

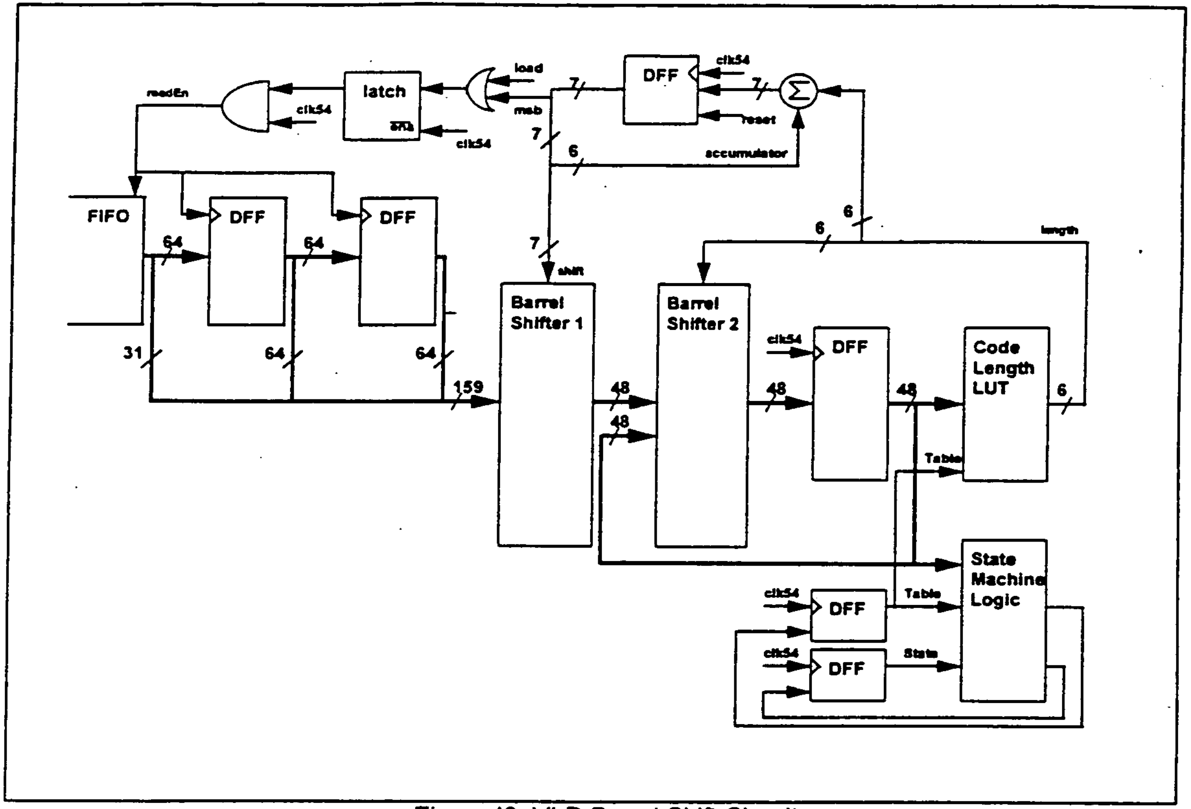


Figure I3: VLD Barrel Shift Circuit

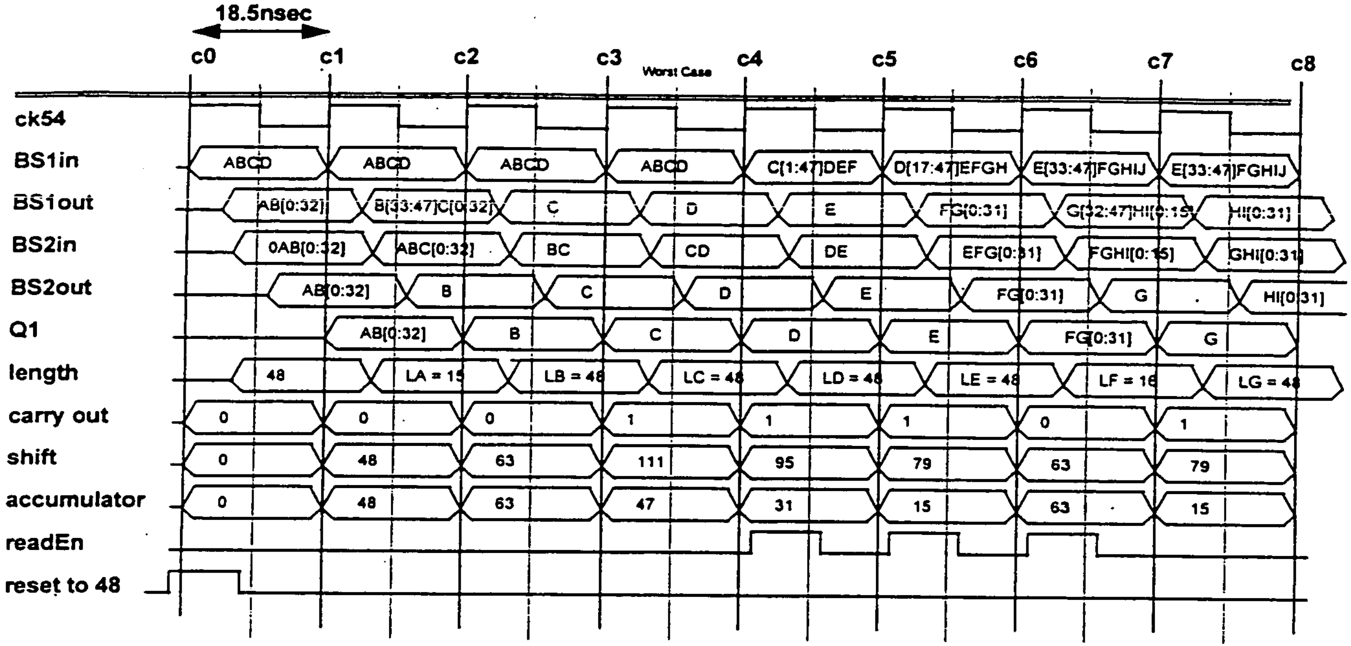
There are two barrel shifters in this circuit. Barrel Shifter 2 aligns the variable length data on codeword boundaries. As each codeword is decoded through the look up tables, the length of that decoded codeword is fed back to Barrel Shifter 2, and that codeword is therefore shifted out of the circuit (i.e. that data is thrown away).

Barrel Shifter 1 is responsible for aligning the new data from the input FIFO stage with the trailing edge of variable length data from Barrel Shifter 1. In most cases this edge of data is not on a codeword boundary. This is best understood by examining the timing diagram of figure I4. For example, notice in clock period c6 where Barrel Shifter 1 aligns the most significant bits ([G[32:47]) with the least significant bits (G[0:31]) from Barrel Shifter 2. (My notation for least significant bit here refers to the first bit of a variable length code.)

The large bus sizes of this circuit are required to handle the worst case codeword length case. The longest codeword length (48 bits) is during the decoding of two AC escape codes in one cycle(e.g. two consecutive 0 run AC escape codes). Therefore Barrel Shifter 2 must output 48 bits to the LUT's. Because it is possible to have two 48 length codewords in a row, the input to Barrel Shifter 2 must be 2*48 = 96 bits wide.

Barrel Shifter 2 must be able to handle worst case combination of codeword lengths. That situation is whenever there is only one remaining unused bit in a 64 bit FIFO word, and then the next 2 codewords are the largest codeword length (48 bits). This is demonstrated in the timing diagram of figure I4. The first two codewords(A,B) add up to 63 bits, and then the next two codewords(C,D) are 48 bit length codewords. In clock cycle c3 the worst case occurs when Barrel

Shifter 1 must shift by an amount of 111, which is the largest shift it can allow since this points to the last 48 bits in the 159 input word. In this case those last 48 bits represent one codeword(D) that we decode in clock cycle c4. Without the large bus widths, we would have to insert wait cycles in order to handle this case.



Codeword lengths = A=15,B=48,C=48,D=48,E=48,F=16,G=48,H=16,I=48,J=16

Figure 14 Barrel Shift Circuit Timing with Worst Case Codeword Lengths

There are three time critical paths in this design. Each one must complete within one clock cycle (18.5 nsec). Critical path 1 is from the 159 input of Barrel Shifter 1 through Barrel Shifter 2 and to the 48 bit register before the LUT's. Critical path 2 is from the 48 bit register through the length LUT, through the 6 bit adder to the 7 bit accumulator register. Critical path 3 starts at the 48 bit register, through the length LUT, through Barrel Shifter 2 and back to the same 48 bit register. In preliminary synthesis and layouts, Critical Path 3 has proven to be the worst of these three paths.

In order to optimize the speed through these paths several techniques are being utilized. First, the barrel shifters are being implemented with muxes composed of tri state buffers(see figure 17). This application guarantees that the output of the barrel shifter will always be driving (i.e. the output is never in tri state).

Second, the barrel shifter architecture was designed to minimize the fan out on each bit, and also to hide as much of the shift transition time on each mux. Figure 15 shows the block diagram of the Barrel Shifter 2 device. Notice that the mux sizes increase as the data flows through the circuit. Therefore, the shift transition time of all 3 mux stages occurs in parallel which results in hiding the transition time of the first two mux stages. Figure 18 shows an approximate timing diagram of this technique.

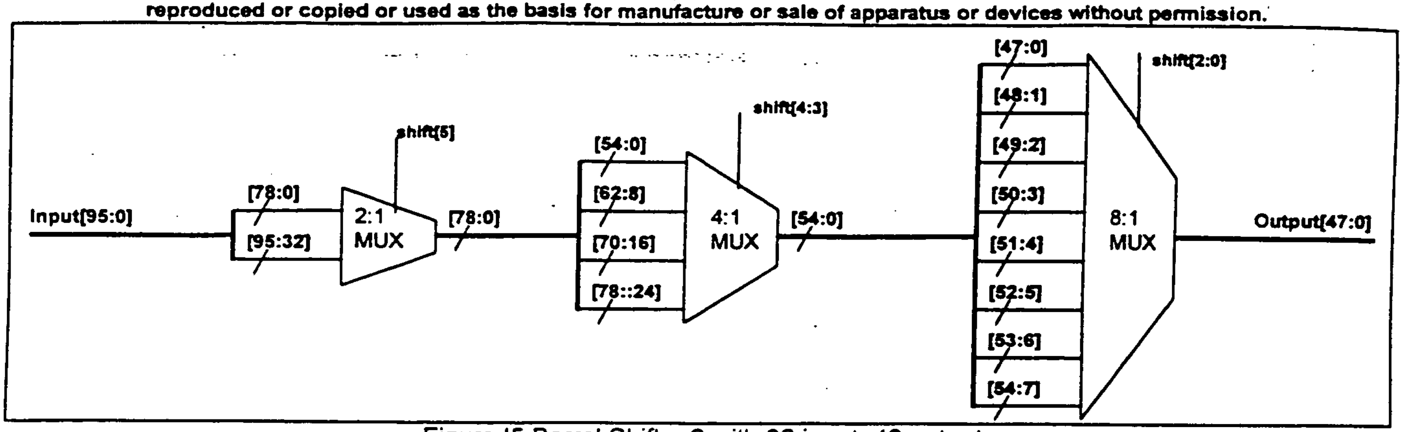


Figure I5 Barrel Shifter 2 with 96 input, 48 output

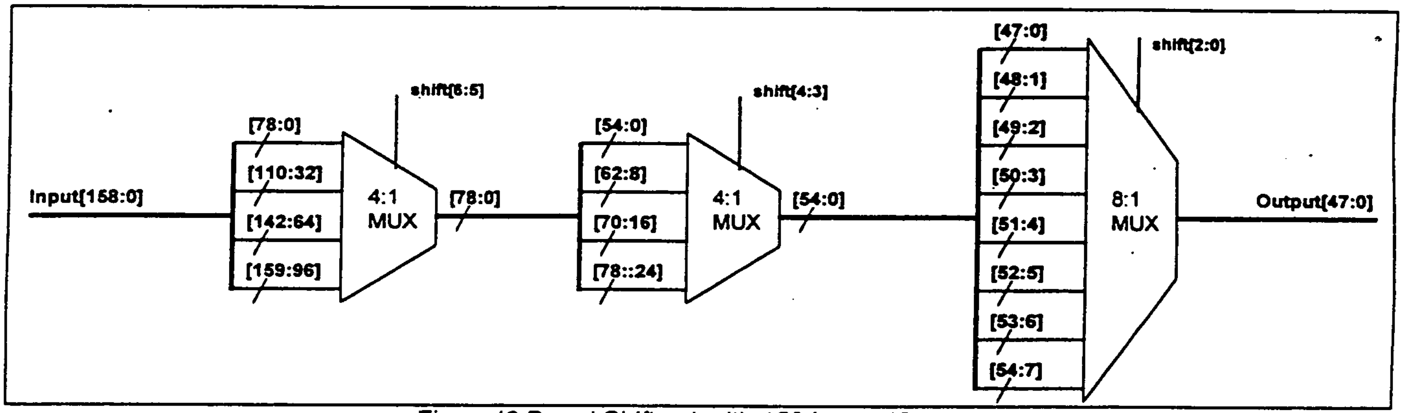


Figure I6 Barrel Shifter 1 with 159 input, 48 output

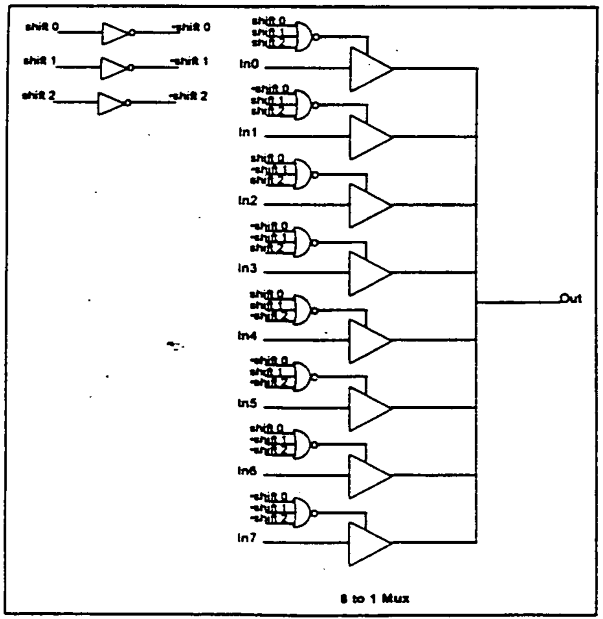


Figure I7: 8 to 1 Mux

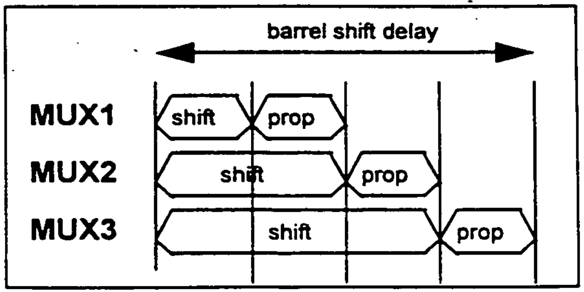


Figure I8: propagation hiding

Third, the length LUT was separated from the value LUT. This greatly reduces the size of the LUT needed in the time critical path since the variable length codes can be grouped into common length categories. The tables below list the common codeword prefixes that are used in the length LUT.

MBA Table:

	prefix								length					
	0	1	2	3	4	5	6	7	dec	4	3	2	1	0
A	1								1	0	0	0	0	1
B	0	1							3	0	0	0	1	1
C	0	0	1						4	0	0	1	0	0
D	0	0	0	1					5	0	0	1	0	1
E	0	0	0	0	1	1			7	0	0	1	1	1
F	0	0	0	0	1	0			8	0	1	0	0	0
G	0	0	0	0	0	1	1		8	0	1	0	0	0
H	0	0	0	0	0	1	0	1	10	0	1	0	1	0
I	0	0	0	0	0	1	0	0	11	0	1	0	1	1
J	0	0	0	0	0	0			11	0	1	0	1	1

MBT-Intra Table:

prefix		length					
	0	dec	4	3	2	1	0
A	0	1	0	0	0	0	1
B	1	2	0	0	0	1	0

MBT-Pred Table:

	prefix					length					
	0	1	2	3	4	dec	4	3	2	1	0
A	1					1	0	0	0	0	1
B	0	1				2	0	0	0	1	0
C	0	0	1			3	0	0	0	1	1
D	0	0	0	1		5	0	0	1	0	1
E	0	0	0	0	1	5	0	0	1	0	1
F	0	0	0	0	0	6	0	0	1	1	0

MBT-Bidir Table:

	prefix				length					
	0	1	2	3	dec	4	3	2	1	0
A	1				2	0	0	0	1	0
B	0	1			3	0	0	0	1	1
C	0	0	1		4	0	0	1	0	0
D	0	0	0	1	5	0	0	1	0	1
E	0	0	0	0	6	0	0	1	1	0

CBP Table:

	prefix						length					
	0	1	2	3	4	5	dec	4	3	2	1	0
A	1	1	1				3	0	0	0	1	1
B	1	1					4	0	0	1	0	0
C	1	0	1				4	0	0	1	0	0
D	1	0	0				5	0	0	1	0	1
E	0	1					5	0	0	1	0	1
F	0	0	1				6	0	0	1	1	0
G	0	0	0	1			8	0	1	0	0	0
H	0	0	0	0	1		8	0	1	0	0	0
I	0	0	0	0	0	1	8	0	1	0	0	0
J	0	0	0	0	0	0	9	0	1	0	0	1

MV Table:

	prefix									length					
	0	1	2	3	4	5	6	7	8	dec	4	3	2	1	0
A	1									1	0	0	0	0	1
B	0	1								3	0	0	0	1	1

C	0	0	1							4	0	0	1	0	0
D	0	0	0	1						5	0	0	1	0	1
E	0	0	0	0	1	1				7	0	0	1	1	1
F	0	0	0	0	1	0				8	0	1	0	0	0
G	0	0	0	0	0	1	1			8	0	1	0	0	0
H	0	0	0	0	0	1	0	1		10	0	1	0	1	0
I	0	0	0	0	0	1	0	0	1	10	0	1	0	1	0
J	0	0	0	0	0	1	0	0	0	11	0	1	0	1	1
K	0	0	0	0	0	0	1	1		11	0	1	0	1	1

DC-Size-Y Table:

	prefix								length					
	0	1	2	3	4	5	6	7	dec	4	3	2	1	0
A	1	0							3	0	0	0	1	1
B	0								2	0	0	0	1	0
C	1	1	0						3	0	0	0	1	1
D	1	1	1	0					4	0	0	1	0	0
E	1	1	1	1	0				5	0	0	1	0	1
F	1	1	1	1	1	0			6	0	0	1	1	0
G	1	1	1	1	1	1	0		7	0	0	1	1	1
H	1	1	1	1	1	1	1	0	8	0	1	0	0	0
I	1	1	1	1	1	1	1	1	9	0	1	0	0	1

DC-Size-C Table:

	prefix									length					
	0	1	2	3	4	5	6	7	8	dec	4	3	2	1	0
A	1	0								2	0	0	0	1	0
B	0									2	0	0	0	1	0
C	1	1	0							3	0	0	0	1	1
D	1	1	1	0						4	0	0	1	0	0
E	1	1	1	1	0					5	0	0	1	0	1
F	1	1	1	1	1	0				6	0	0	1	1	0
G	1	1	1	1	1	1	0			7	0	0	1	1	1
H	1	1	1	1	1	1	1	0		8	0	1	0	0	0
I	1	1	1	1	1	1	1	1	0	9	0	1	0	0	1
J	1	1	1	1	1	1	1	1	1	10	0	1	0	1	0

DCT-AC1 Table:

	prefix												length					
	0	1	2	3	4	5	6	7	8	9	10	11	dec	4	3	2	1	0
A	1	0											2	0	0	0	1	0
B	1	1											3	0	0	0	1	1
C	0	1	1										4	0	0	1	0	0
D	0	1	0										5	0	0	1	0	1
E	0	0	1	0	1								6	0	0	1	1	0
F	0	0	1	1									6	0	0	1	1	0
G	0	0	0	1									7	0	0	1	1	1
H	0	0	0	0	1								8	0	1	0	0	0
I	0	0	1	0	0								9	0	1	0	0	1
J	0	0	0	0	0	0	1						11	0	1	0	1	1
A	0	0	0	0	0	0	0	1					13	0	1	1	0	1
B	0	0	0	0	0	0	0	0	1				14	0	1	1	0	0
C	0	0	0	0	0	0	0	0	0	1			15	0	1	1	1	1
D	0	0	0	0	0	0	0	0	0	0	1		16	1	0	0	0	0
E	0	0	0	0	0	0	0	0	0	0	0	1	17	1	0	0	0	1
F	0	0	0	0	0	1							24	1	1	0	0	0

DCT-AC2 Table:

	prefix												length					
	0	1	2	3	4	5	6	7	8	9	10	11	dec	4	3	2	1	0
A	1	0											3	0	0	0	1	1
B	0	1	0										4	0	0	1	0	0
C	1	1	0										4	0	0	1	0	0

D	0	1	1	0									4	0	0	1	0	0
E	0	1	1	1									5	0	0	1	0	1
F	0	0	1	1									6	0	0	1	1	0
G	0	0	1	0	1								6	0	0	1	1	0
H	1	1	1	0									6	0	0	1	1	0
I	0	0	0	1									7	0	0	1	1	1
J	0	0	0	0	1								8	0	1	0	0	0
K	1	1	1	1	0								8	0	1	0	0	0
L	1	1	1	1	1	0	0						8	0	1	0	0	0
M	0	0	1	0	0								9	0	1	0	0	1
N	1	1	1	1	1	1							9	0	1	0	0	1
O	0	0	0	0	0	0	1	0					10	0	1	0	1	0
P	0	0	0	0	0	0	1	1	1				10	0	1	0	1	0
Q	0	0	0	0	0	0	1	1	0				11	0	1	0	1	1
R	1	1	1	1	1	0	1						9	0	1	0	0	1
S	0	0	0	0	0	0	0	1					13	0	1	1	0	1
T	0	0	0	0	0	0	0	0	1				14	0	1	1	0	0
U	0	0	0	0	0	0	0	0	0	1			15	0	1	1	1	1
V	0	0	0	0	0	0	0	0	0	0	1		16	1	0	0	0	0
W	0	0	0	0	0	0	0	0	0	0	0	1	17	1	0	0	0	1
X	0	0	0	0	0	1							24	1	1	0	0	0

DCT-AC1 Zero run Length Table:

	vlc prefix															length					
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	dec	4	3	2	1	0
1	1	1														3	0	0	0	1	1
2	0	1	0	0												5	0	0	1	0	1
3	0	0	1	0	1											6	0	0	1	1	0
4	0	0	0	0	1	1	0									8	0	1	0	0	0
5	0	0	1	0	0	1	1	0								9	0	1	0	0	1
6	0	0	1	0	0	0	0	1								9	0	1	0	0	1
7	0	0	0	0	0	0	1	0	1	0						11	0	1	0	1	1
8	0	0	0	0	0	0	0	1	1	1	0	1				13	0	1	1	0	1
9	0	0	0	0	0	0	0	1	1	0	0	0				13	0	1	1	0	1
10	0	0	0	0	0	0	0	1	0	0	1	1				13	0	1	1	0	1
11	0	0	0	0	0	0	0	1	0	0	0	0				13	0	1	1	0	1
12	0	0	0	0	0	0	0	0	1	1	0	1	0			14	0	1	1	1	0
13	0	0	0	0	0	0	0	0	1	1	0	0				14	0	1	1	1	0
14	0	0	0	0	0	0	0	0	1	0	1	1	1			14	0	1	1	1	0
15	0	0	0	0	0	0	0	0	0	1						15	0	1	1	1	1
16	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	16	1	0	0	0	0
17	0	0	0	0	0	0	0	0	0	0	1	0				16	1	0	0	0	0
18	0	0	0	0	0	1	0	0	0	0	0	0				24	1	1	0	0	0

DCT-AC1 One run Length Table:

	vlc prefix															length					
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	dec	4	3	2	1	0
1	0	1	1													4	0	0	1	0	0
2	0	0	0	1	1	0										7	0	0	1	1	1
3	0	0	1	0	0	1	0	1								9	0	1	0	0	1
4	0	0	0	0	0	0	1	1	0	0						11	0	1	0	1	1
5	0	0	0	0	0	0	0	1	1	0	1	1				13	0	1	1	0	1
6	0	0	0	0	0	0	0	0	1	0	1	1	0			14	0	1	1	1	0
7	0	0	0	0	0	0	0	0	1	0	1	0	1			14	0	1	1	1	0
8	0	0	0	0	0	0	0	0	0	0	1	1	1			16	1	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1	0	1		16	1	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	16	1	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	1	0	0		17	1	0	0	0	1
12	0	0	0	0	0	1	0	0	0	0	0	1				24	1	1	0	0	0

DCT-AC2 Zero run Length Table:

	vlc prefix															length					
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	dec	4	3	2	1	0
1	1	0														3	0	0	0	1	1
2	1	1	0													4	0	0	1	0	0
3	0	1	1	1												5	0	0	1	0	1
4	1	1	1	0												6	0	0	1	1	0
5	0	0	0	1	0											7	0	0	1	1	1
6	1	1	1	1	0	1	1									8	0	1	0	0	0
7	1	1	1	1	1	0	0									8	0	1	0	0	0
8	0	0	1	0	0	0	1									9	0	1	0	0	1
9	1	1	1	1	1	0	1									9	0	1	0	0	1
10	1	1	1	1	1	1	1									9	0	1	0	0	1
11	0	0	0	0	0	0	0	0	0	1						15	0	1	1	1	1
12	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	16	1	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	1	0				16	1	0	0	0	0
14	0	0	0	0	0	1	0	0	0	0	0	0				24	1	1	0	0	0

DCT-AC2 One run Length Table:

	vlc prefix															length					
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	dec	4	3	2	1	0
1	0	1	0													4	0	0	1	0	0
2	0	0	1	1	0											6	0	0	1	1	0
3	1	1	1	1	0	0	1									8	0	1	0	0	0
4	0	0	1	0	0	1	1	1								8	0	1	0	0	0
5	0	0	1	0	0	0	0	0								9	0	1	0	0	1
6	0	0	0	0	0	0	0	0	1	0	1	1	0			14	0	1	1	1	0
7	0	0	0	0	0	0	0	0	1	0	1	0	1			14	0	1	1	1	0
8	0	0	0	0	0	0	0	0	0	0	1	1	1			16	1	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1	0	1		16	1	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	16	1	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	1	0	0		17	1	0	0	0	1
12	0	0	0	0	0	1	0	0	0	0	0	1				24	1	1	0	0	0

Figure I9 shows the top level view of the length LUT circuit. Each MPEG table is represented by a different sub-LUT. For example, the MBA LUT is a separate table. The state machine determines which table will be used for the next codeword and selects the correct table with the table address bus.

The AC coefficient table is more complicated since it must decode consecutive codewords with runs of 0.0 and 0,1 in one clock cycle. This is achieved by separating the LUT into a run 0,0 table, a run 0,1 table and a normal table (i.e. all AC codewords). The run 0,0 LUT is designed to do the look ahead for the second 0 run codeword in parallel to the look up for the first 0 run codeword(This technique is faster than a serial technique which would require the second codeword table to wait for the first codeword length to be decoded.) The run 0,0 table is formed by repeating the DCT-AC1 run 0 table at every possible 0 run offset. That is, in order to form all the possible combinations of 0,0 runs, the table must look for a 0 run codeword with offset 0, and a 0 run codeword at all other possible offsets (for the DCT-AC1 table there are 11 possible 0 run lengths or offsets). If a 0,0 run is found, then the r00 signal will become true and control the tri state buffer (see figure I10). This technique is also used for the run 0,1 table.

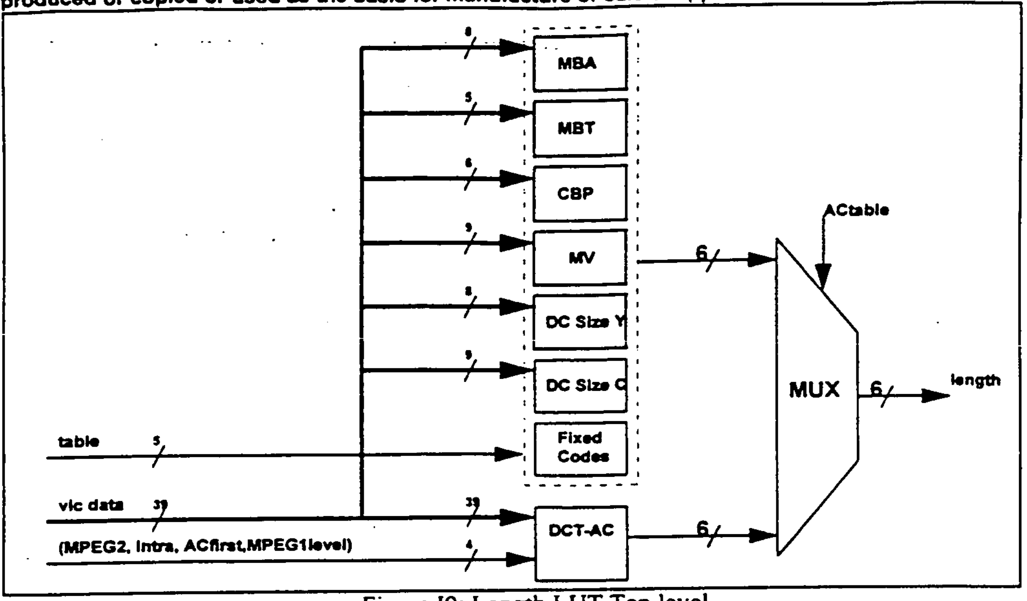


Figure I9: Length LUT Top level

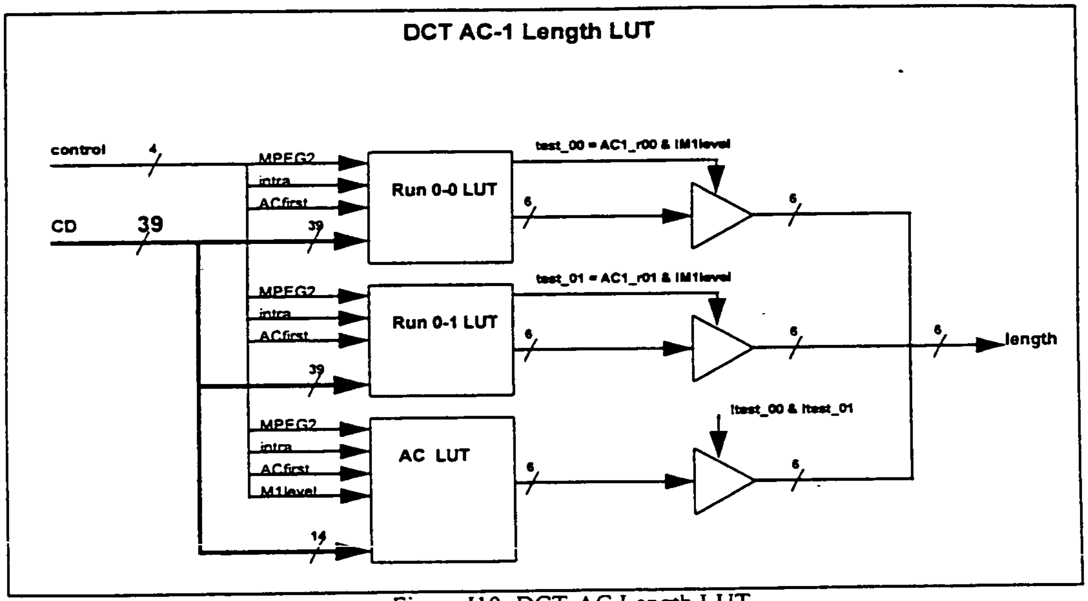


Figure I10: DCT-AC Length LUT

The value LUT is a pipelined architecture since it is not in the time critical path. There are two paths through the Value decode pipeline(see figure I11). First, the top path decodes all codewords including all AC codewords. The second path looks at the decoded value from the top table. If the codeword decoded was a 0 run coeficient, then the bit stream is shifted to the next codeword boundary (with a 46 input 24 output barrel shifter). The bitstream is then examined for a consecutive 0 run or 1 run codeword. This second table is only composed of the 0 run and 1 run coeficients. The value of this second 0 run or 1 run code is decoded and sent to the lower Block FIFO.

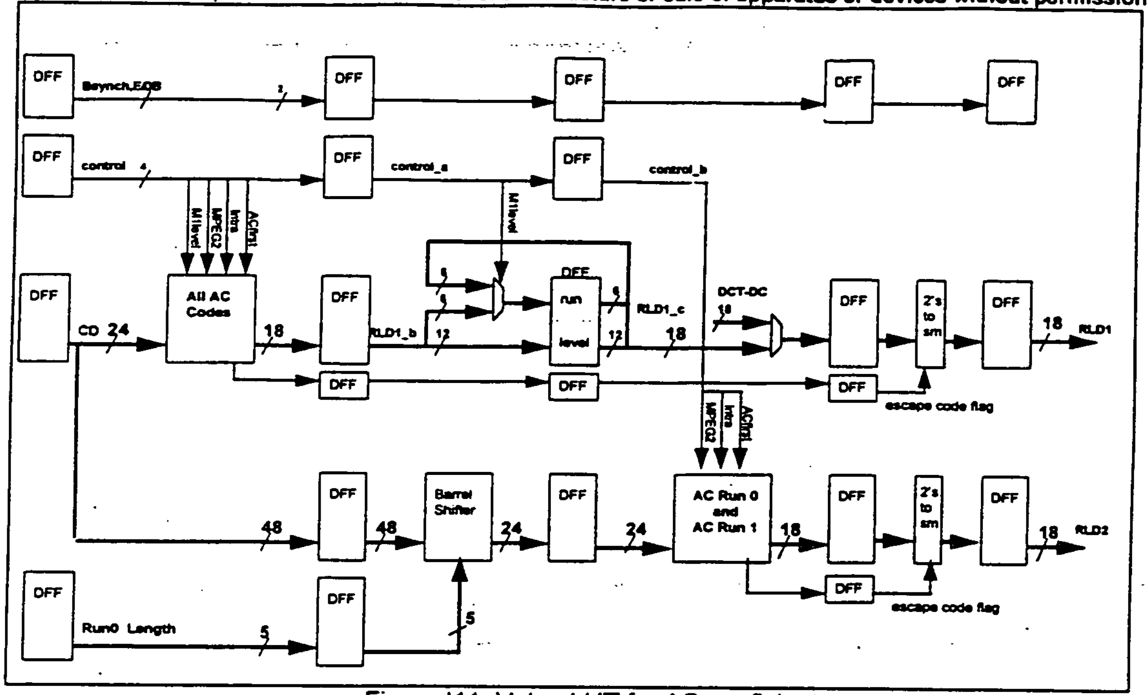


Figure I11: Value LUT for AC coeficients

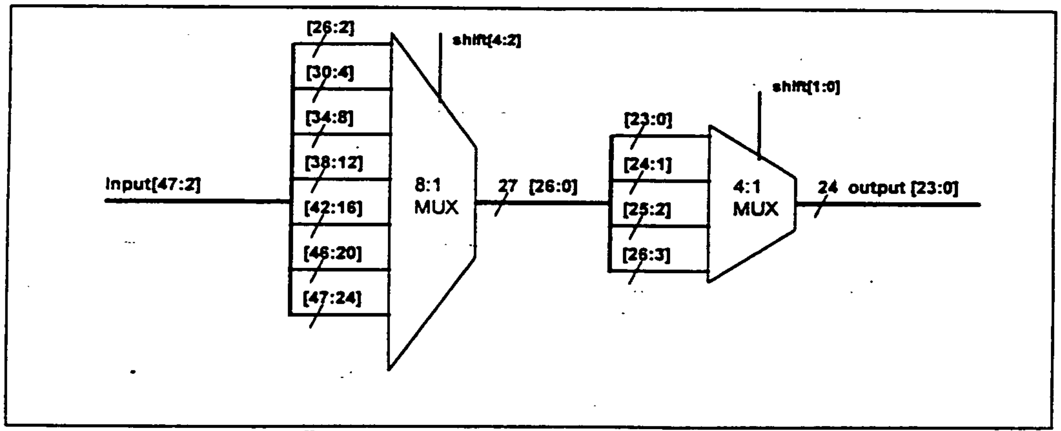


Figure I12 : Barrel Shifter with 46 input : 24 output

I4. VLD STATE MACHINES

The state machine that controls the decoding process of the VLD is separated into several sub state machines and one Master state machine. Figure I13 shows the block diagram of this organization.

The Master state machine is responsible for determining the next type of codeword in the bit stream. That is, it must select the correct VLC table in order to decode the next codeword. It uses inputs from the immediate data in the bit stream plus data that was transmitted earlier in the bit stream (Sequence layer, GOP layer, Picture layer,Slice layer, and MB layer). The Sequence, GOP and Picture layer data is parsed by the external micro processor and available through registers to the Master state machine. The macroblock type, motion vector data, and coded block pattern are all parsed by the value look up table and latched for later use by the Master state machine.

The macroblock address decoding is done in parallel to the master state machine. As macroblock address data is parsed by the master state machine, the GO signal increments the MBA count. The MBA unit then determines if skip macroblocks should be inserted. (NOTE: The MBA unit from the 3520 may be re-used for this design. It still needs to be determined if the circuit can run at the required speed.)

The Master state machine uses two smaller state machines to decode the block layer data and the motion vector data. Since these types of codewords usually occur multiple time per macroblock, this saves duplicated states in the state machine.

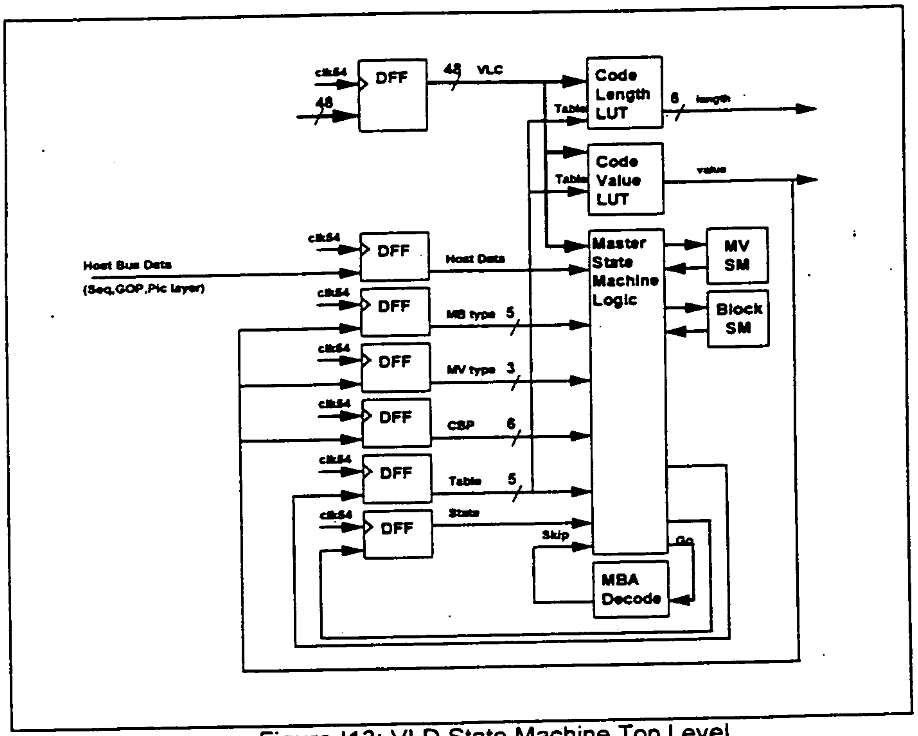


Figure I13: VLD State Machine Top Level

I5. MV UNIT

The motion vector unit converts the motion vector data that is decoded in the value look up table, and converts that into true motion vectors for use in the motion compensation unit. MPEG motion vectors are differentially encoded from one macroblock to the next. The MV unit must follow the MPEG rules for decoding motion vectors in all cases (i.e. dual prime, field predictors, error concealment Intra vectors, etc.) These rules are described in the MPEG2 specification section 7.6.

The MV unit from the Sti3520 will be re-used for this design with some control changes. In the Sti3520 design, the MV unit worked in series with the decoding path. This architecture requires that the MV unit run at the same clock as the coefficient decoding clock. However, the needed decode rate for motion vectors is much lower; maximum 4 motion vectors per macroblock. Therefore, by using the MV unit as a parallel task machine, it can run at much slower rates. For the HD-VLD we plan to run it at its STi3520 designed rate (27 Mhz). The control of the MV unit will change in order to achieve this, however the implemented algorithms specific to MPEG MV decoding will not change.

I6. DCT-DC DECODING

DC coeficients for Intra macroblocks and non-Intra macroblocks are coded differently. In the case of non-Intra macroblocks, the DC value is coded in the run length look up tables B-14. For Intra macroblocks the DC values are differentially encoded. When the codeword components for the Intra DC component exit the Value LUT, then the DCT-DC deocde unit decodes these into real DC values. All other codewords by pass the DCT-DC decode unit(figure I14)

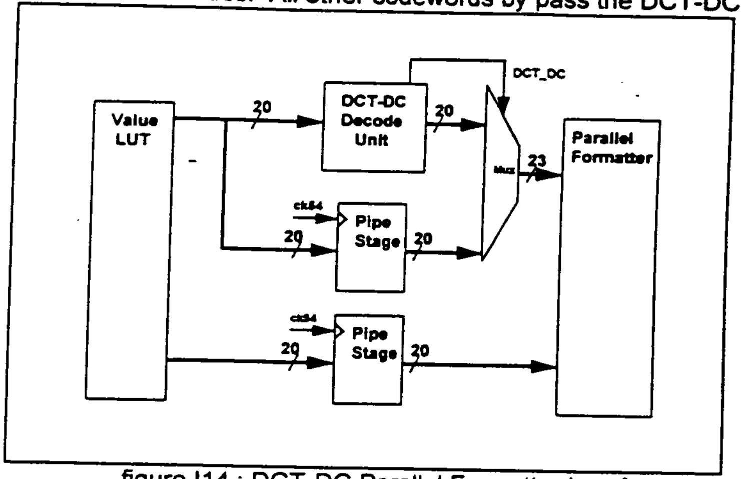


figure I14 : DCT-DC Parallel Formatter Interface

For each Intra DC coeficient, a differential value is transmitted in the bitstream. This value is transmitted using the codewords dct_dc_size_luminance, dct_dc_size_chrominance, and dct_dc_differential. If the differential value between two successive Intra DC coefficients is 0 then only the dct_dc_size component is sent (i.e. it is equal to 0: no differential value). When the differential value is other than 0 then the dct_dc_size codeword denotes the number of bits transmitted for the dct_dc_differential.

The dct_dc_differential is the delta between successive Intra DC coefficients. A separate predictor is kept for the Y, Cr, and Cb component. This predictor will be reset to mid range (based on intra_dc_precision) for the following events:

- the start of a slice
- when a non-intra macroblock is decoded
- when a skipped macroblock is decoded

The hardware that will implement this function will be a state machine that watches for the DC values decoded from the value LUT. The predictors will be formed and stored in registers. See figure I15

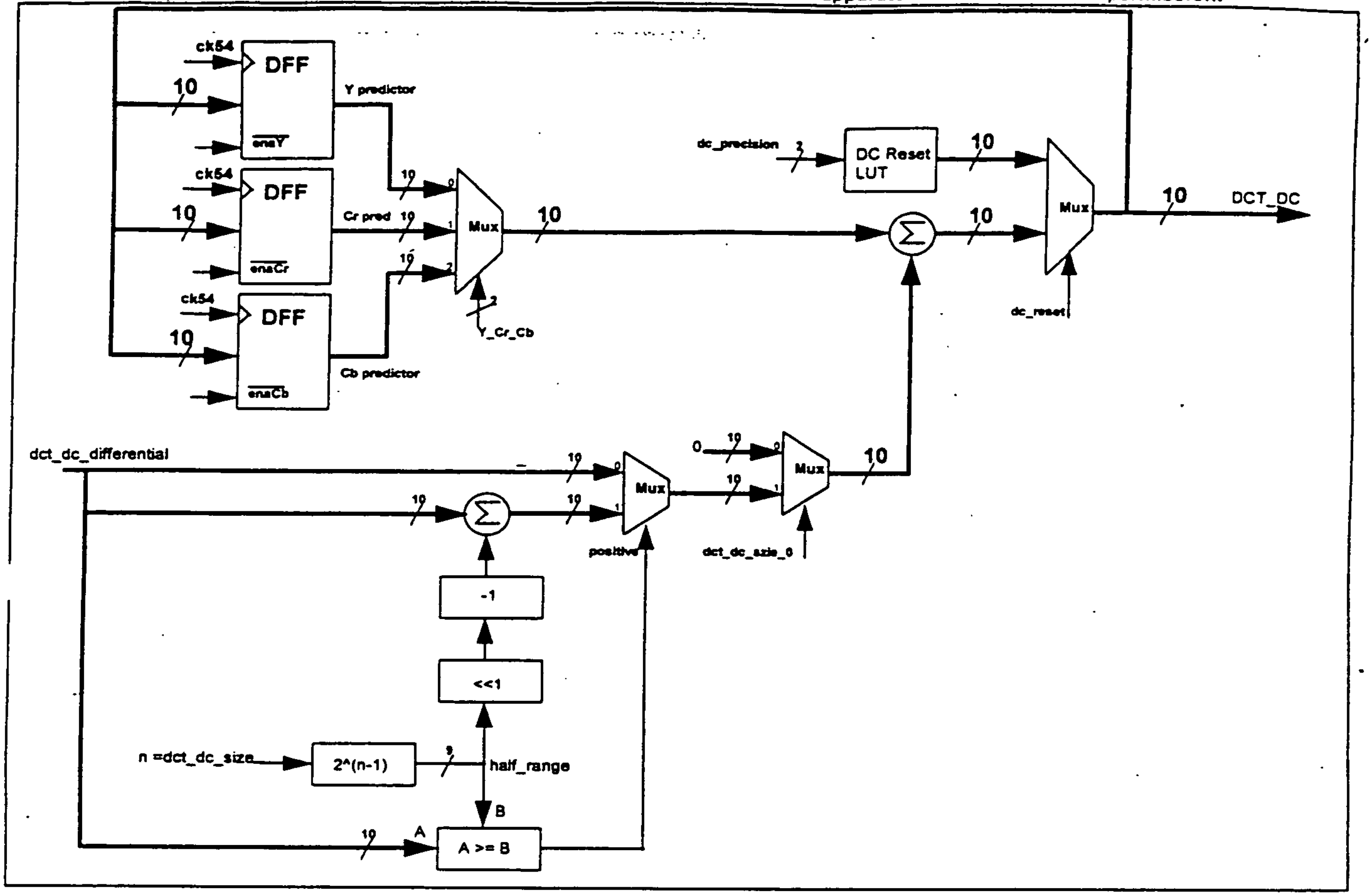


figure I15 : DCT DC Decoding

I7. PARALLEL FORMATTER

The parallel formatter is the interface to the 2 pipeline FIFO's. Each FIFO will hold a minimum of one macroblock. The data from the output of the value LUT alternately transmitted between these two output FIFO's. The dd macroblocks are transmitted to the top pipeline and the even macroblocks are transmitted to the lower pipeline (including skipped macroblocks).

Because the VLD can produce two codewords in one 54 MHz clock cycle, its effective peak codeword rate is 108 M codewords/sec. This implies that either the pipeline FIFO must be capable of running at a 108 MHz rate, or that the pipeline FIFO is two words wide running at 54 MHz. This later approach has been taken due to the physical limitations of the FIFO's in this library.

Data output from the value LUT represents 1 or 2 codewords per cycle. If this data was transmitted to the 2 word wide FIFO, then there would be many empty (or wasted) memory locations in the FIFO(and therefore the FIFO would have to be twice as big as necessary). The parallel formatter packs this data into the FIFO so that there is no wasted space in the FIFO. As data flows through the circuit, one of 12 possible states will occur(see figure I16). The parallel formatter attempts to always write two codewords into the FIFO.

At the end of each block it is possible that there is only one codeword left to write. In this case, then the parallel formatter inserts a dummy codeword in one position of the FIFO. The run length decoder input interface will have to detect this dummy codeword and throw it away. The state machine truth table for the parallel formatter is shown below.

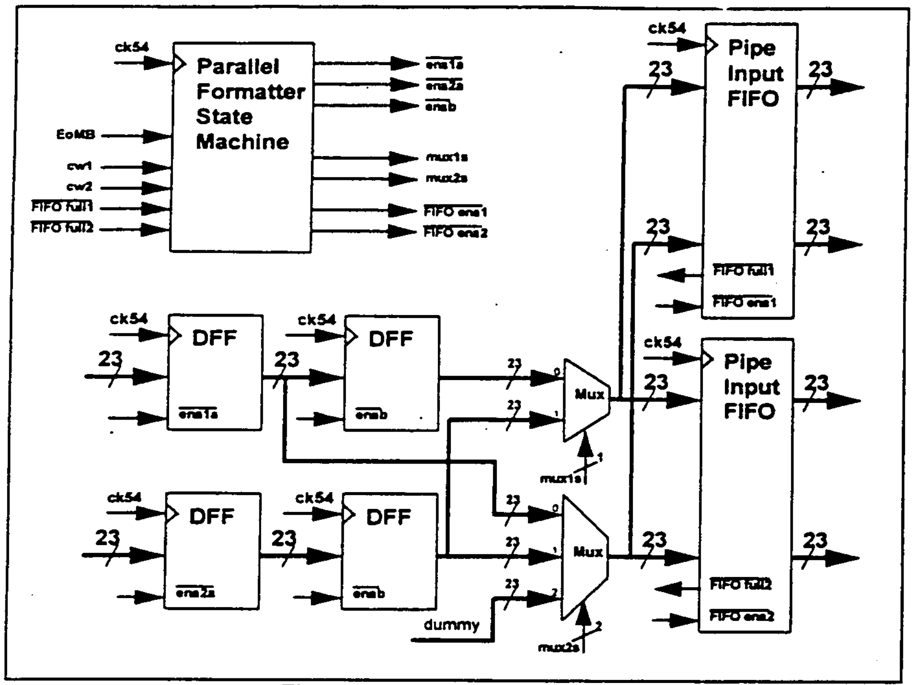


Figure 116: Parallel Formatter

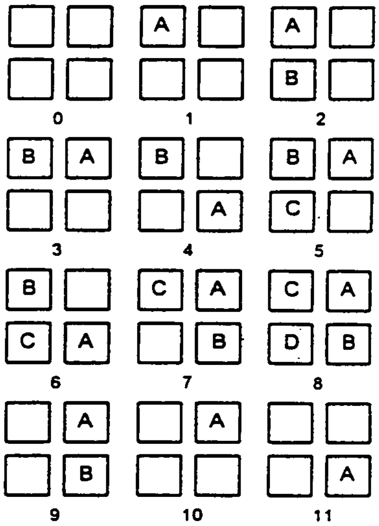


Figure 117: Parallel Formatter States

Parallel Formatter State Machine Truth Table

Inputs			Outputs			
State	CW1	CW2	Mux-1s	Mux-2s	FIFOena	next state
0	0	0	X	X	1	0
0	1	0	X	X	1	1
0	1	1	X	X	1	2
1	0	0	0	2	0	10
1	1	0	0	0	0	3
1	1	1	0	0	0	5
2	0	0	0	1	0	9
2	1	0	0	1	0	7
2	1	1	0	1	0	8
3	0	0	X	X	1	0
3	1	0	X	X	1	1
3	1	[illegible]	X	X	1	2
4	0	0	X	X	1	0
4	1	0	X	X	1	1
4	1	1	X	X	1	2
5	0	0	1	2	0	11
5	1	0	1	0	0	4
5	1	1	1	0	0	6
6	0	0	1	2	0	11
6	1	0	1	0	0	4
6	1	1	1	0	0	6
7	0	0	0	2	0	10
7	1	0	0	0	0	3
7	1	1	0	0	0	5
8	0	0	0	1	0	9

8	1	0	0	1	0	7
8	1	1	0	1	0	8
[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]
10	0	0	X	X	1	0
10	1	0	X	X	1	1
10	1	1	X	X	1	2
[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]

I8. ERROR CONCEALMENT PIPELINE INTERFACE

An adjacent MBT/MV FIFO is used for error concealment. When an error is detected in the VLD, then the currently decoded block is discarded from the pipeline Block FIFO, and then fake macroblocks are generated that use the macroblock type and motion vectors from the top adjacent macroblock. In the case where there is more than one motion vector, only one is saved in the FIFO and used for concealment.

Because there are two pipelines, when an error occurs in the VLD, it must alternate fake macroblocks from one .peline to the next. Therefore, one common FIFO controller will be designed for both pipeline interfaces. This controller will have some duplicated functionality for each pipeline, but a common error mechanism for concealment (see figure I18). The common MBT/MV delay line ensures that the correct error concealment motion vector will be used for any missing macroblocks(regardless of image size).

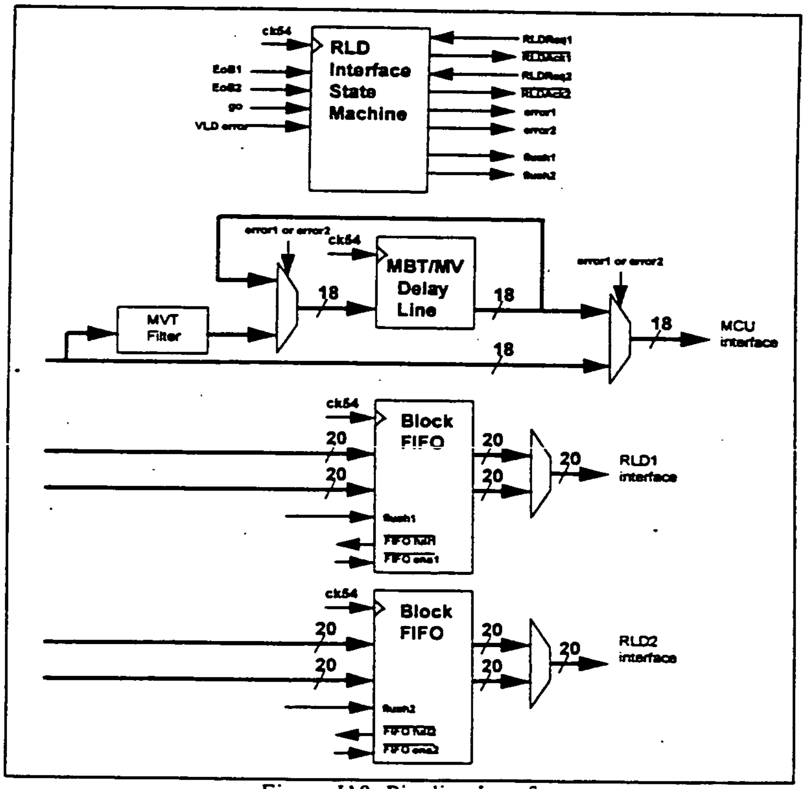


Figure I18: Pipeline Interface

HD MPEG VIDEO DECODER

APPENDIX J

DECODE PIPE

Thomson Consumer Electronics, Inc.
Indianapolis, Indiana, USA

Revision No. 2.0

J1. OVERVIEW

This document describes the overwiew schematic of the HD MPEG IC decoding operator and the interfaces between the internal blocks. The decoding operator includes the decoding pipelines, the motion compensation unit, the compression/decompression block and the fifos interfacing with the external memory.

J2. INTERFACES

J2.1. VLD to pipe interface : 20 bits + 1 acknowledge signal

Level vector : 12 bits
Run vector : 6 bits
EOB signal (end of block)
BSync signal (block synchronization)

J2.2. VLD to MCU and IQ interface : 20 bits

Go: 1 bit
Data bus : 14 bits
Address bus : 4 bits
acknowledge signal

MCU extract vectors and macroblock type information from this interface, IQ information type on macroblock (intra or not intra) and reodering fifos also (field or frame macroblock, i.e DCT field flag, when frame structure picture).

Semantic and syntax :

1st information : MBtype - vbusA = 'hC - vbusD[4:0] has the following meaning
data[4] = DCTfield
data[3:0] = 1 : intra macroblock
data[3:0] = 2 : forward progressive (frame structure) or forward 16x16 (field structure) predicted macroblock
data[3:0] = 3 : forward field (frame structure) or forward 16x8 (field structure) predicted macroblock
data[3:0] = 4 : backward progressive (frame structure) or backward 16x16 (field structure) predicted macroblock
data[3:0] = 5 : backward field (frame structure) or backward 16x8 (field structure) predicted macroblock
data[3:0] = 6 : bidir progressive (frame structure) or bidir 16x16 (field structure) predicted macroblock
data[3:0] = 7 : bidir field (frame structure) or bidir 16x8 (field structure) predicted macroblock
data[3:0] = 8 : copy predicted macroblock = type 2 macroblock with "0" motion vectors (for skipped macroblocks in predicted frames)

data[3:0] = 9 : copy bidirectional macroblock (same type as preceeding macroblock but in progressive mode, motion vectors are resent by the VLD, for skipped macroblocks in bidirectional pictures)

2nd information : motion vectors (as needed)
vbusA = 0 : data = forward horizontal vector (top field if field prediction or upper blocks if 16x8)
vbusA = 1 : data = forward vertical vector (top field if field prediction or upper blocks if 16x8)
vbusA = 2 : data = forward horizontal vector bottom field or lower blocks
vbusA = 3 : data = forward vertical vector bottom field or lower blocks
vbusA = 4 : data = backward horizontal vector (top field if field prediction or upper blocks if 16x8)
vbusA = 5 : data = backward vertical vector (top field if field prediction or upper blocks if 16x8)
vbusA = 6 : data = backward horizontal vector bottom field or lower blocks
vbusA = 7 : data = backward vertical vector bottom field or lower blocks
vbusD[11:0] = half pel vector (two's complement)
vbusD[12] : for vertical only if vbusD[13] = 1 : field selection of prediction (from MVFS)
vbusD[13] : for vertical vectors only : 1 if field prediction (i.e. vector in the field)

3rd information : GO signal - validate all preceeding macroblock information and allow the MCU to start.

J2.3. IDCT to ordering fifos interface

Each IDCT outputs 9 bit (two's complement from -255 to +255) per cycle at 54MHz. This interface should not be stopped by the rest of the operating part through the dimensioning of the rest of the pipe.

J2.4. Adder to compression unit

The adder receive 1 pixel (error pixel) from the pipeline, one from the MCU (prediction pixel) and send the resut to the compression unit or formatting block of the reconstruction fifo. The scan order of one unit is the same on all interfaces : horizontal scanning of the blocks, interleaved blocks, in fact suitable order for compression unit. The frequency of this interface is 54 MHz. The adder is a strictly pipeline block with one or two cycles.

The control waits for the MCU and compression (or reconstruction if no compression) request to be set to assert a request to the pipelinen fifos. When the pipeline acknowledges, this ack is transmitted to the other blocks with the correct delay. During the decoding of a picture, the interfaces of the adder should not be stop often (around 13 % of the time).

J2.5. Compression units or Adders to formatting block of the reconstruction fifos

Depending on compression or no compression, the compression blocks and its interfaces are shunted. So the acknowledges signals can come from the compression units or the adders, same for requests signals.

The formatting block and fifos receive data from both pipeline and merge it into the fifos to prepare the bursts access to the external SDRAM. The pipeline are smoothly synchronised through this interface, as at their input through the VLD interface and through the output of the MCU. In other words, there is a minimum and maximum difference of state between both pipelines, that is only constrained by the output fifos of the VLD, the output of the MCU and the reconstruction fifos.

In case of H/2-M/2 mode, the behaviour is different : blocks for pipe 1 and pipe 2 are stuck together and horizontally decimated before going to the same compression unit. In this case, only compression unit 1 is used, and pipes are synchronized at the input of this unit.

The data input scan order is the same than the output order of the reordering buffers for each pipe, except that out of the compression, blocks are deinterleaved. Two separate 8 bit bus are send from the compression unit, each bus concerning one block.

J2.6. Decompression block and MCU buffers Interfaces

The motion compensation unit controls the input DBus buffers and formatting buffers. The input of a single decompression block is two 32 bit words of alternate blocks, each block scanned horizontally in a way suitable for direct decompression.

The MCU controller controls all fifo buffers of this part of the pipe : input fifos forward and backwards, reodering buffers forward and backwards and output buffers. The decompression units are 4 in each prediction direction (F or B), plus one shared between both direction, and every decompression units has its own interface :

two 32-bits bus input, to allow interleaved decompression : each bus represent data of one block (block a or block b). Each block is scanned horizontally in a way suitable for direct decompression (it is the same order than the output of the compression unit)

one 32-bits bus output (representing 4 pixels horizontally). This bus outputs two interleaved blocks in horizontal scanning. Two 32-bits words will be available every 8 cycles, one for each block (a and b).

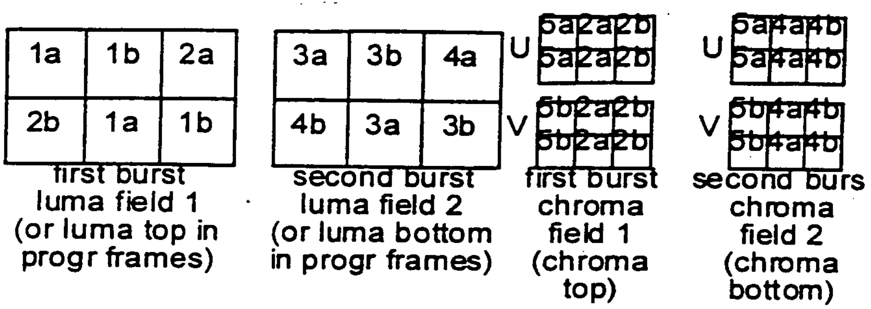
request bus for a new input word : req[1] for block a (the first in the pipeline), req[0] for block b
Acknowledge signal : to answer the valid request

Decompression blocks run at 81 MHz and MCU at 54 MHz. The output interface of the input fifos run at 81 MHz. The reordering buffers make the asynchronous interface between decompression or input fifos and MCU.

The order in wich the blocks are sent to the decompression units is controlled by the MCU.

The next figure represent this order for a single prediction (forward or backward). The ordering is the same in the different modes :

**field store
frame prediction**



The squares are the compressed blocks, the number represents the decompression unit, the letter the interleaved block. The global order for one unit is :

Luma then Chroma
forward then backward for unit 5
burst 1 then burst 2
top to bottom and left to right
The selection and formatting blocks discard data that are not necessary to fill a 20x18 pixel buffer for luma and 12x10x2 of chroma that will be read in appropriate order for filtering (horizontal scanning).

LMC interfaces of reconstruction and MCU

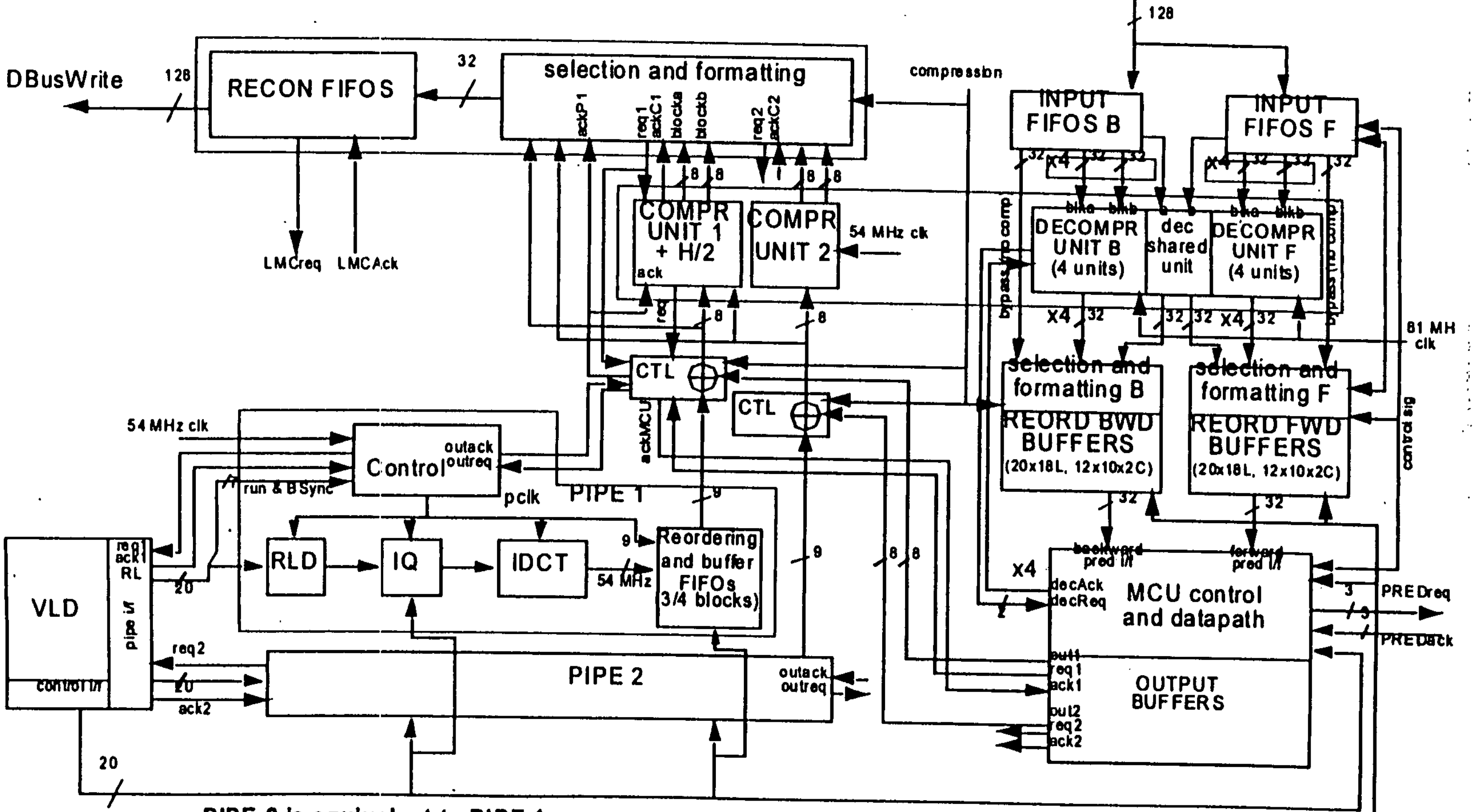
For the reconstruction, LMCreq rises when enough data is present in the fifos to do a burst, depending on the compression mode.

For the MCU unit, the request bus has the following meaning :
PRDreq = 0 : no request
PRDreq = 1 : request for intra prediction access
PRDreq = 2 : request for forward progressive prediction access (frame structure picture) or forward 16x16 prediction access (field structure picture)
PRDreq = 3 : request for forward field prediction access (frame structure picture) or forward 16x8 prediction access (field structure picture)
PRDreq = 4 : request for backward progressive prediction access (frame structure picture) or backward 16x16 prediction access (field structure picture)
PRDreq = 5 : request for backward field prediction access (frame structure picture) or backward 16x8 prediction access (field structure picture)
PRDreq = 6 : request for bidir progressive prediction access (frame structure picture) or bidir 16x16 prediction access (field structure picture)
PRDreq = 7 : request for bidir field prediction access (frame structure picture) or bidir 16x8 prediction access (field structure picture)

LMC acknowledges are :
intraAck : information pulse
forwardAck : forward prediction data access
backwardAck : backward prediction data access
Request from the filter rises when : information from the macroblock to process has been received by the MCU (from the VLD) and place is available on the input fifos for data of a memory burst access.

J3. BLOCK DIAGRAM

Confidential SGS-Thomson Microelectronics



PIPE 2 is equivalent to PIPE 1
only control signals of PIPE 1 are detailed

HD MPEG VIDEO DECODER

APPENDIX K

DIAGRAMS – MEMORY, TOP, ACCESS

Thomson Consumer Electronics, Inc.
Indianapolis, Indiana, USA

Revision No. 2.1

TCE PROPRIETARY AND CONFIDENTIAL

K1. MEMORY CONFIGURATIONS

External memory shall be constructed from SDRAMs or SGRAMS. For the 32Mbit memory configuration, SGRAMS are used. For both 64Mbit and 128Mbit configurations, SDRAMS are used.

K1.1. Memory Interface Description

Below is a list of the pins comprising the external memory interface, along with a brief description of each pin's function.

M_ClkOut (output)	This signal drives the clock for the external SDRAM/SGRAM
M_ClkIn (input)	This input is driven by M_ClkOut to match internal and external memory clock delays
M_ClkEn (output)	Used to place external SDRAM/SGRAM in reduced power mode
M_nRAS (output) **M_nCAS** (output) **M_nWE** (output)	Used for memory access as in conventional DRAM, as well as specifying commands to the SDRAM/SGRAM
M_nCS[1:0] (output)	Chip selects driven by the LMC, allowing multiple external memory configurations
M_Addr[11:0] (output)	Row/column address. Bank select is also implemented via this bus as specified in programmable register ???
M_Data[63:0] (input/output)	Data input/output

K1.2. External Memory Interface Reset

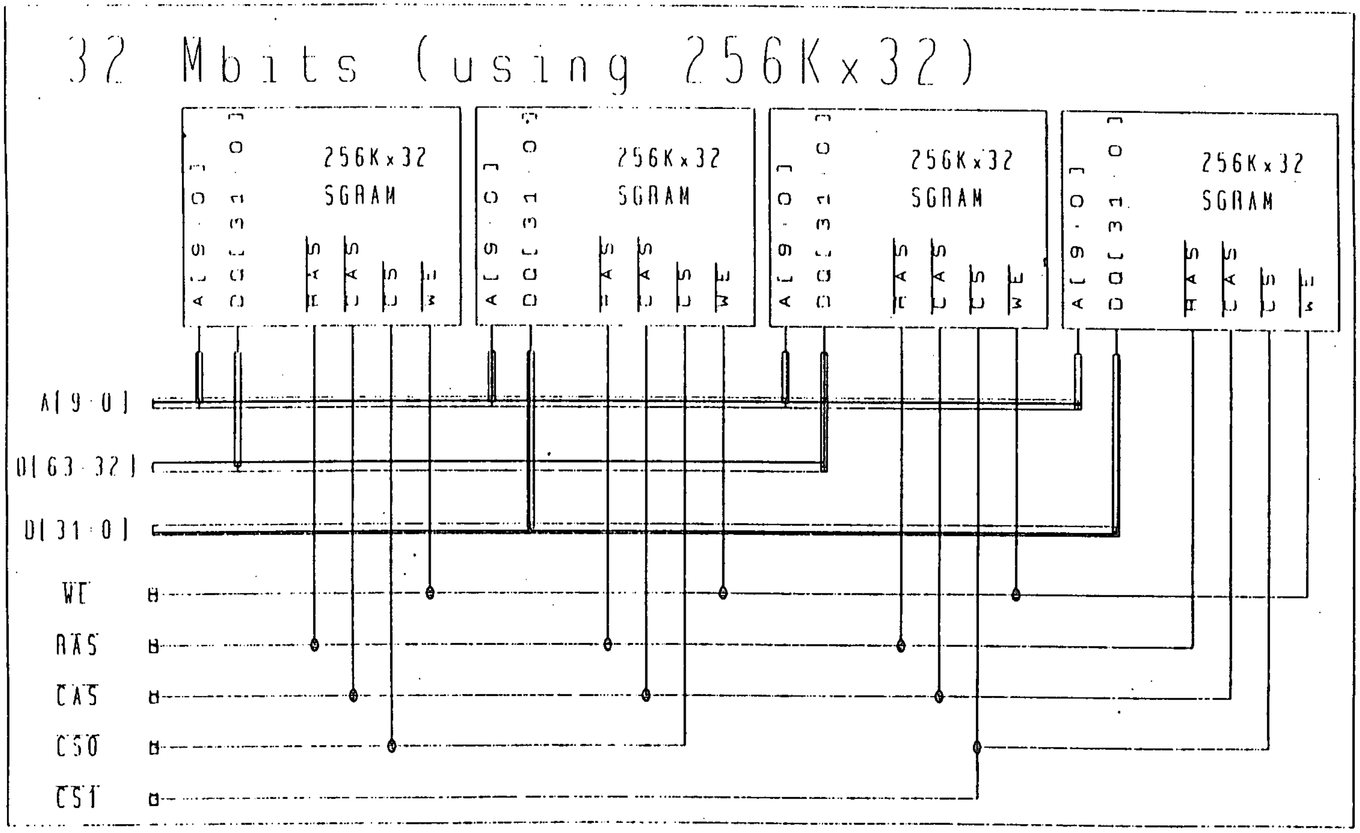
In order to avoid contention on the external Memory interface when reset occurs, the following operations are performed to place the SDRAM/SGRAM in reduced-power mode:

1. The MPEG IC must send a burst stop command to the DRAM (one clock cycle).
2. The MPEG IC deasserts M_ClkEn on the following clock cycle.

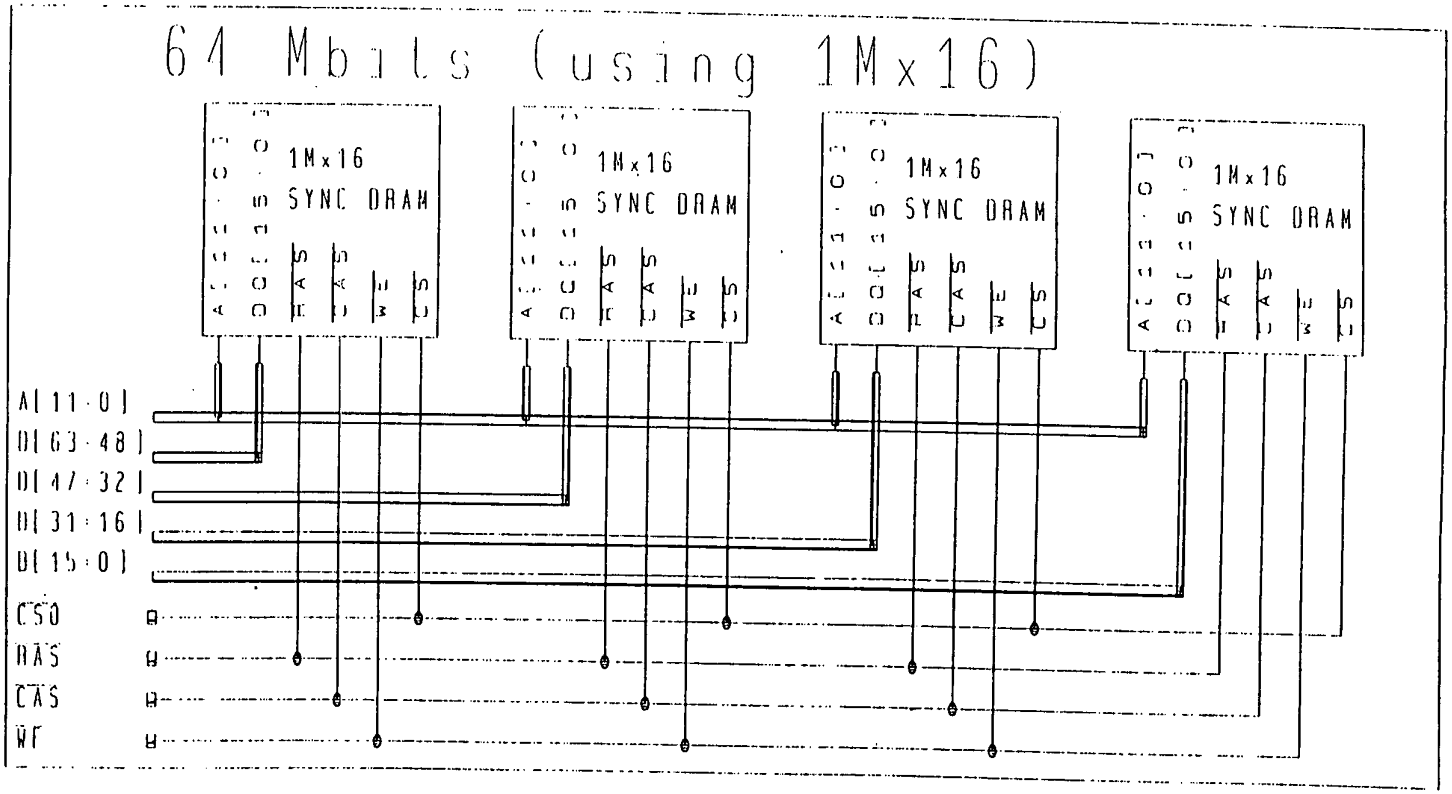
The remainder of the memory interface pins are driven by the HD-MPEG IC to their reduced-power mode values (all zeroes?).

During powerup and normal operation M_ClkEn is asserted.

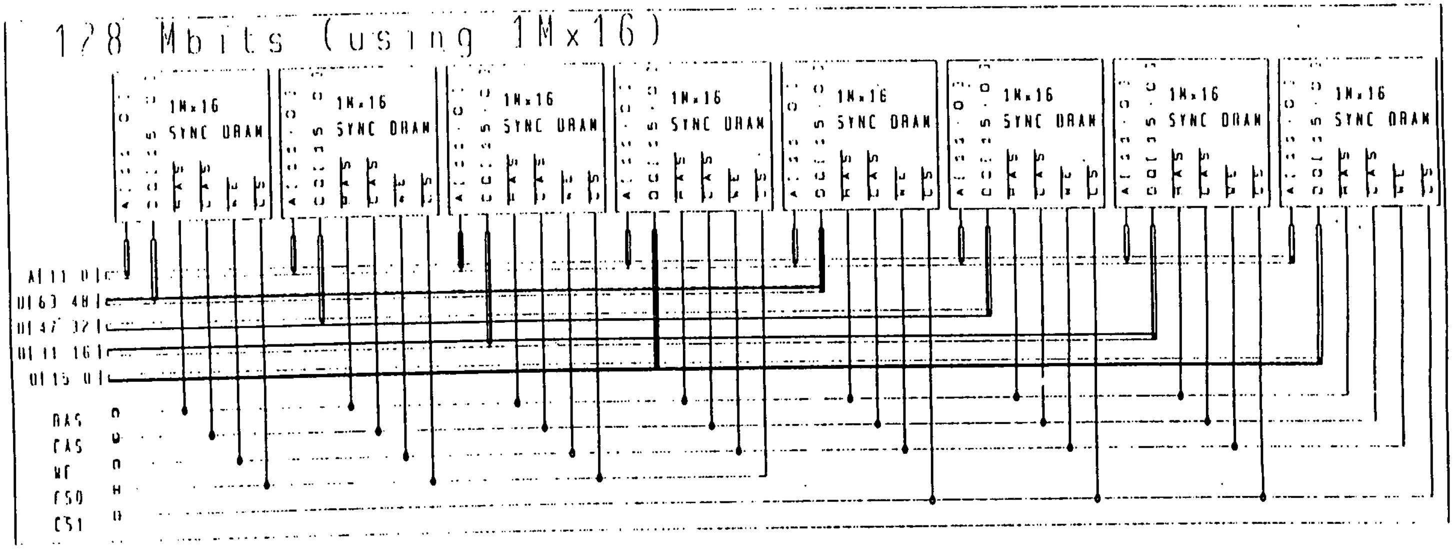
K1.3. SGRAM in 32M bit Configuration



K1.4. SDRAM in 64M bit Configuration

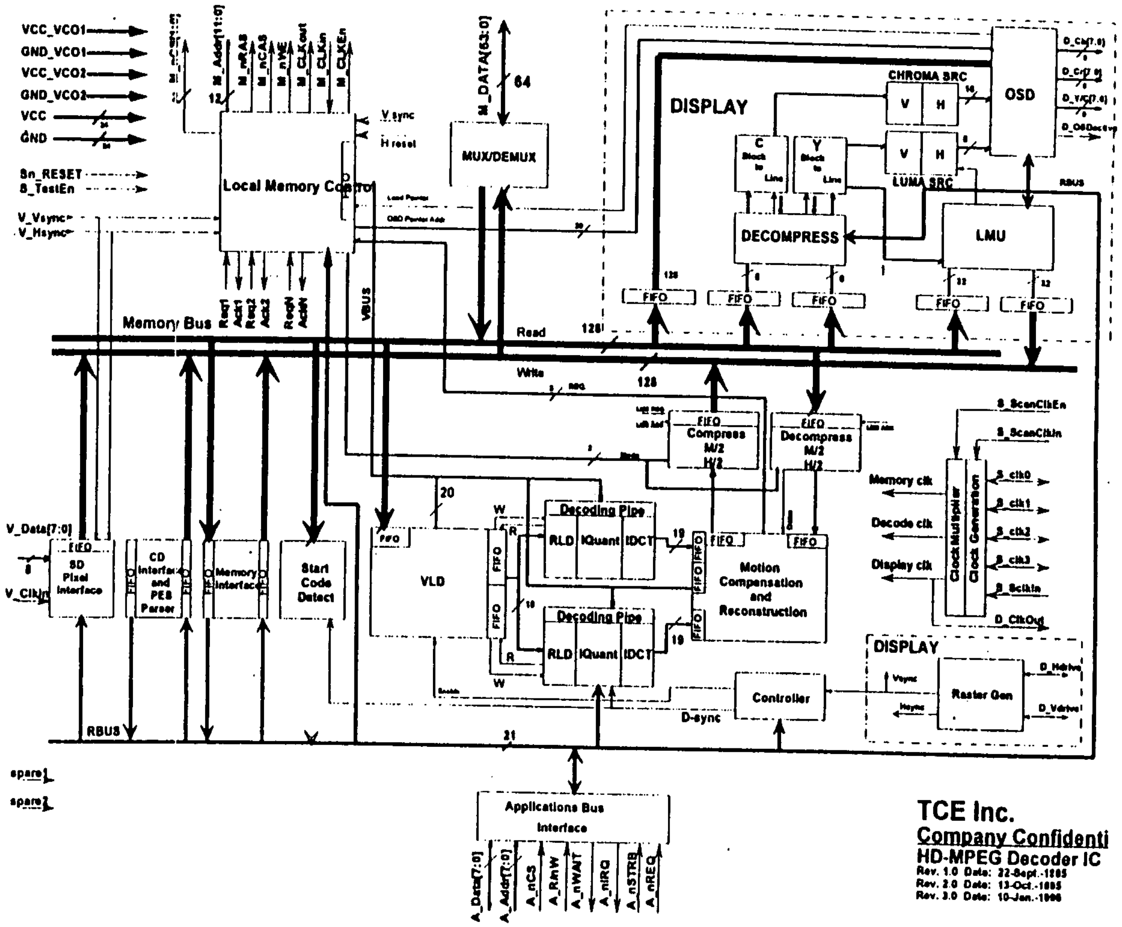


K1.5. SDRAM in 128M bit Configuration



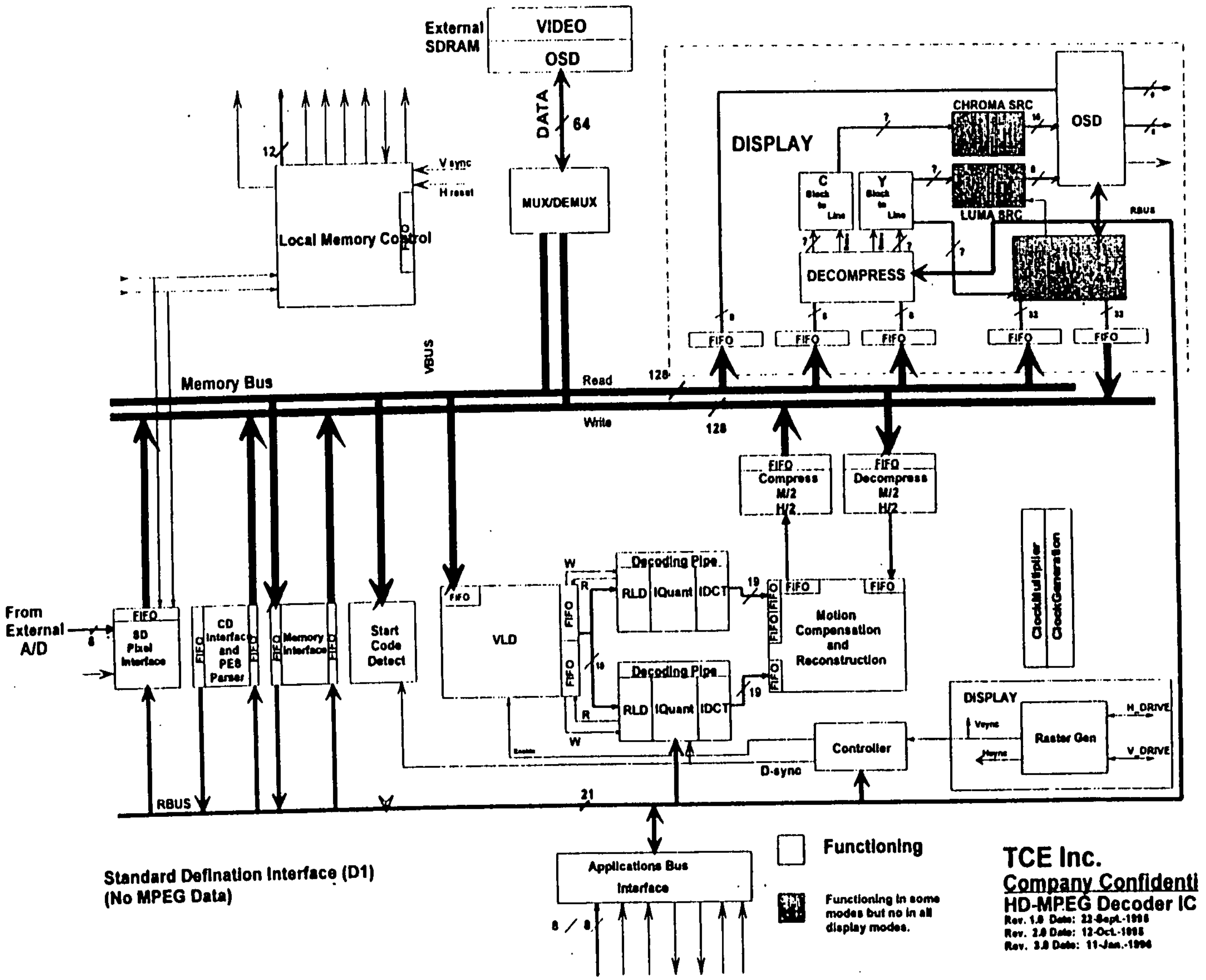
K2. TOP LEVEL DIAGRAMS

K2.1. General Block Diagram

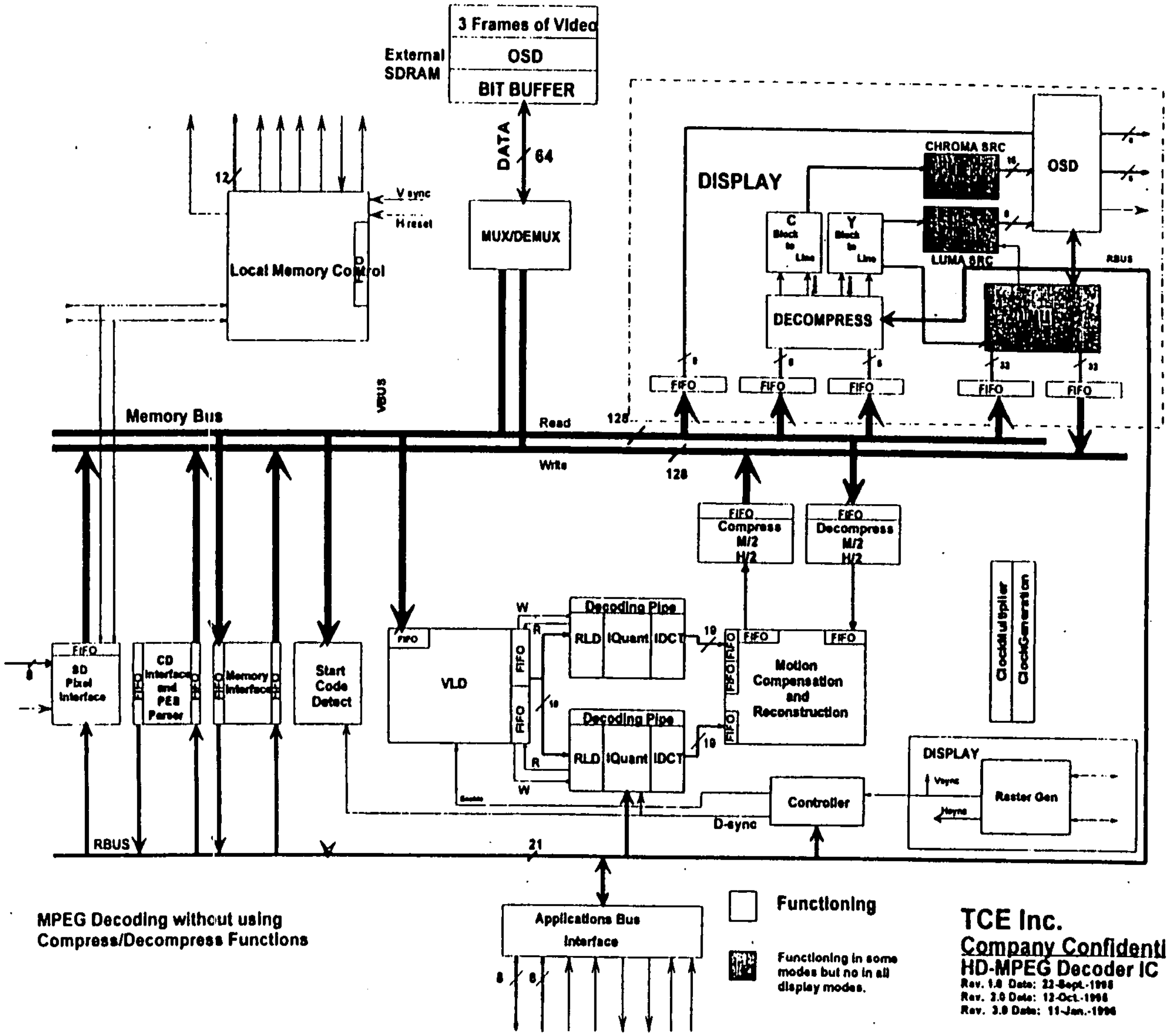


TCE Inc.
Company Confidenti
HD-MPEG Decoder IC
Rev. 1.0 Date: 22-Sept.-1995
Rev. 2.0 Date: 13-Oct.-1995
Rev. 3.0 Date: 10-Jan.-1996

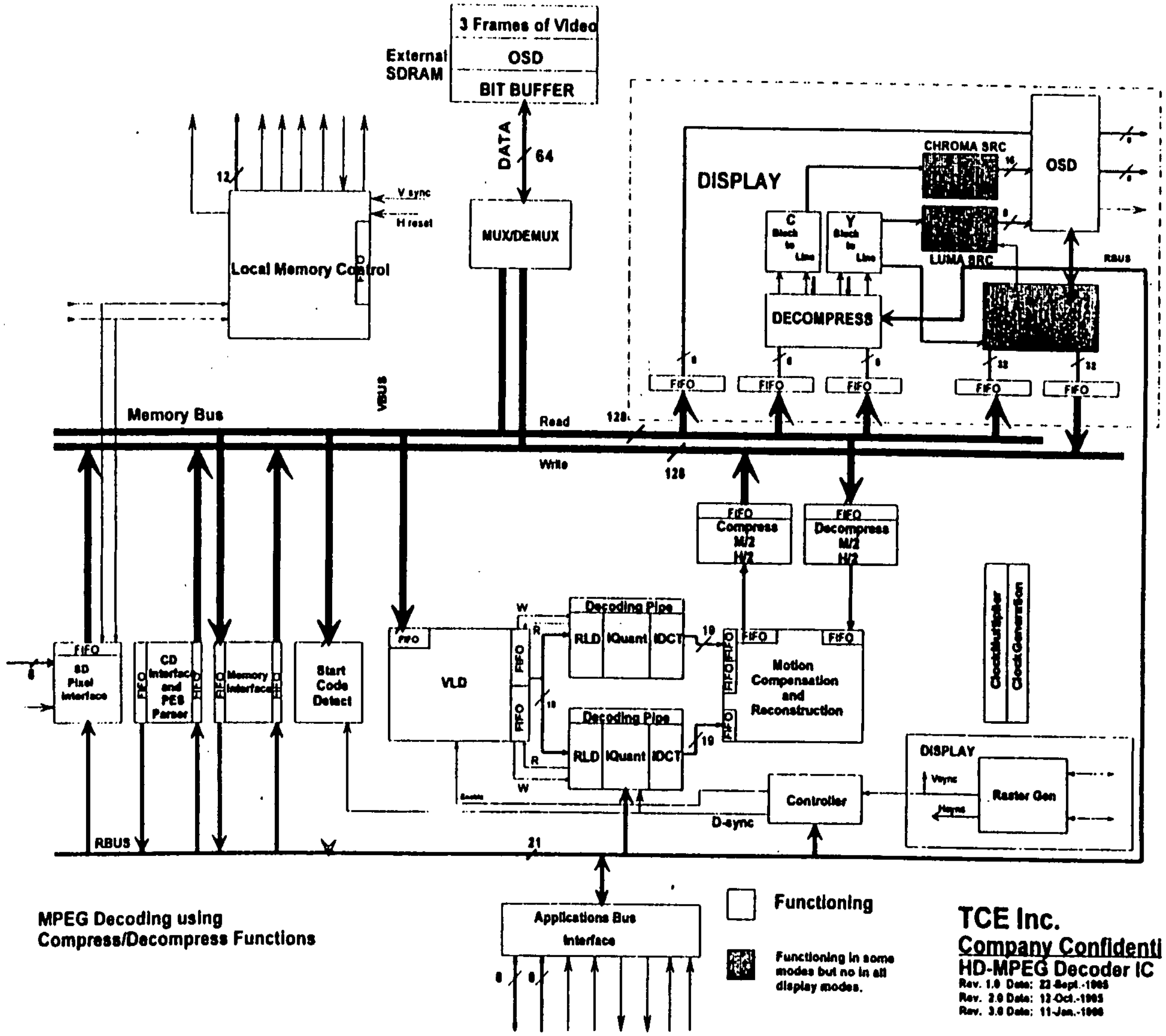
K2.2. D1 Input


External SDRAM
VIDEO
OSD
DATA
64
MUX/DEMUX
12
Vsync
H reset
Local Memory Control
VBUS
Memory Bus
Read
Write
128
128
DISPLAY
CHROMA SRC
LUMA SRC
C Block to Line
Y Block to Line
DECOMPRESS
OSD
RBUS
FIFO
32
FIFO Compress M/2 H/2
FIFO Decompress M/2 H/2
ClockMultiplier
ClockGeneration
From External A/D
8
FIFO
SD Pixel Interface
CD Interface and PES Parser
Memory Interface
Start Code Detect
VLD
W
R
Decoding Pipe
RLD
IQuant
IDCT
19
Motion Compensation and Reconstruction
DISPLAY
Vsync
Hsync
Raster Gen
H_DRIVE
V_DRIVE
Controller
D-sync
RBUS
21
Applications Bus Interface
8 / 8
Standard Defination Interface (D1)
(No MPEG Data)
Functioning
Functioning in some modes but no in all display modes.
TCE Inc.
Company Confidenti
HD-MPEG Decoder IC
Rev. 1.0 Date: 22-Sept.-1995
Rev. 2.0 Date: 12-Oct.-1995
Rev. 3.0 Date: 11-Jan.-1996

K2.3. MPEG Decoding without Compress/Decompress

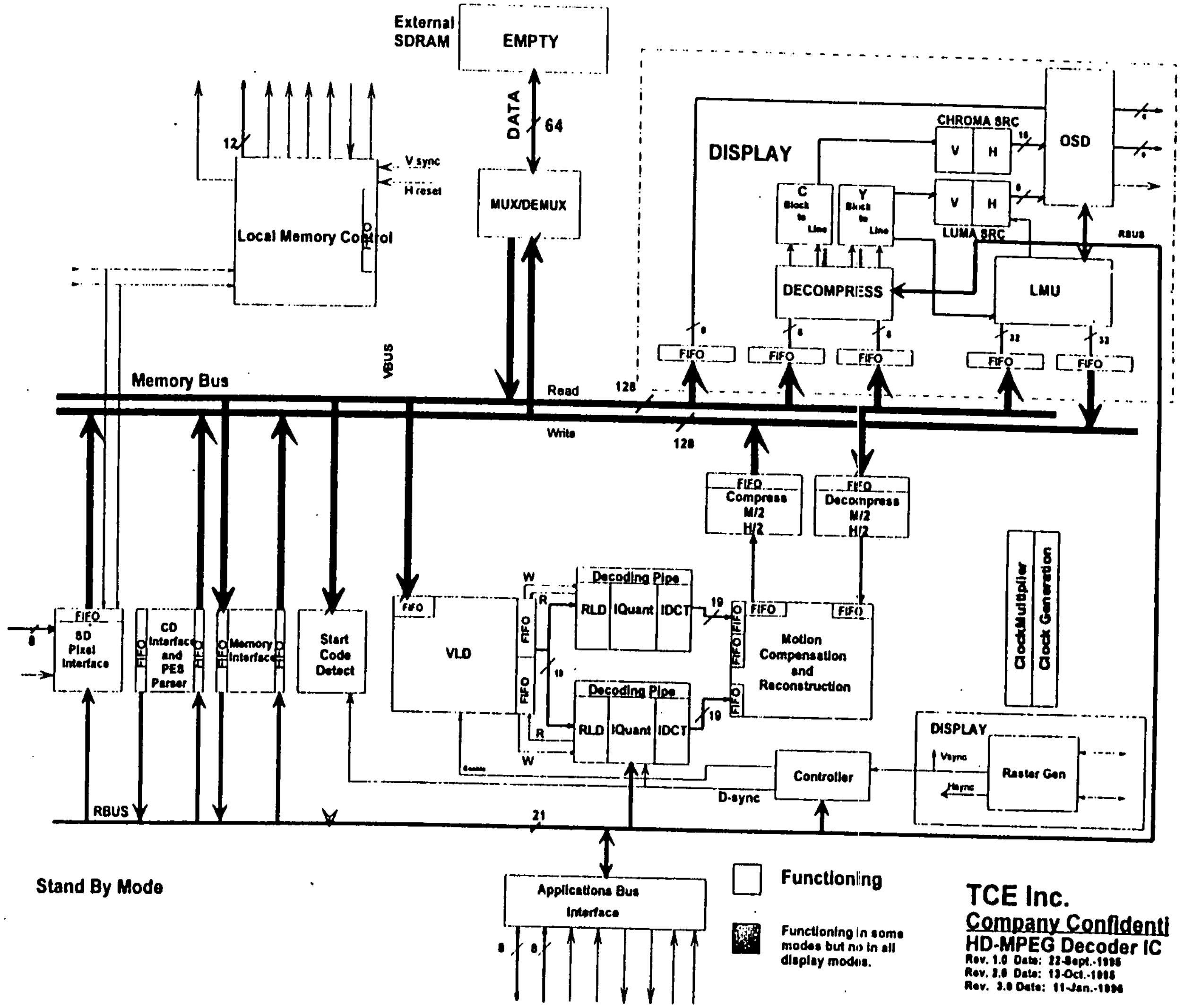


K2.4. MPEG Decoding using Compress/Decompress



MPEG Decoding using Compress/Decompress Functions

K2.5. Standby Mode



Stand By Mode

TCE Inc.
Company Confidentl
HD-MPEG Decoder IC
Rev. 1.0 Date: 22-Sept.-1995
Rev. 2.0 Date: 13-Oct.-1995
Rev. 3.0 Date: 11-Jan.-1996

K3. SDRAM/SGRAM DATA ACCESS DIAGRAMS

K3.1. Introduction

This document shows in detail the SDRAM command and data transfer for the different memory accesses descibed in the local memory controller design specification.

K3.2. Memory Timing Diagrams

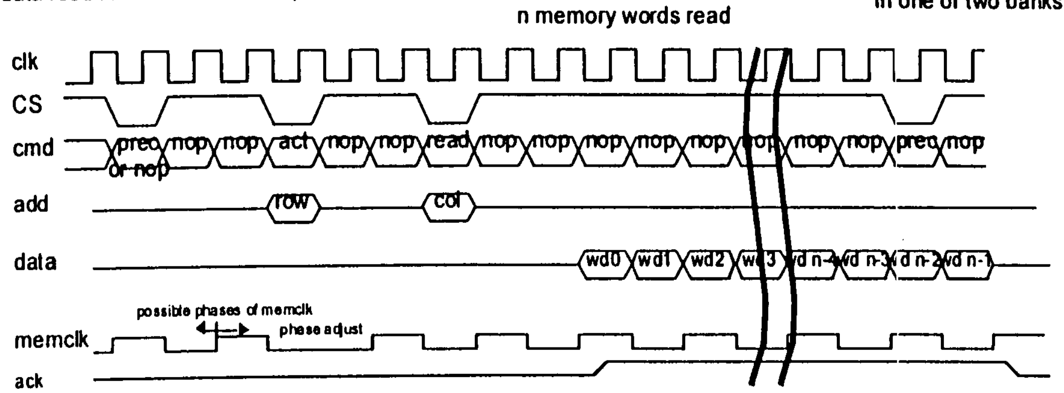
Standard access for read - read access of chroma for display block to line conversion in H/2-M/2 mode.
2 word access, read or write - read access for display of progressive picture in no compression mode
Standard access for write - write access of chroma for reconstruction of field structure picture in M/2-H/2 mode
prediction access in standard mode, column luma with 1 word first bank, 8 second bank, column chroma with 1 word first bank, 4 second bank
prediction access in standard mode, column luma with 2 word first bank, 7 second bank, column chroma with 2 word first bank, 3 second bank
prediction access in standard mode, column luma with 3 word first bank, 6 second bank, column chroma with 3 word first bank, 2 second bank
prediction access in standard mode, column luma with 4 word first bank, 5 second bank, column chroma with 4 word first bank, 1 second bank
prediction access in standard mode, column luma with 5 word first bank, 4 second bank, and with 6 word first bank, 3 second bank
prediction access in standard mode, column luma with 7 word first bank, 2 second bank, and with 8 word first bank, 1 second bank
prediction access in 2M/3 mode
prediction access in H/2-M/2 mode
Example diagram write following read operation

K3.2.1. Standard access for read

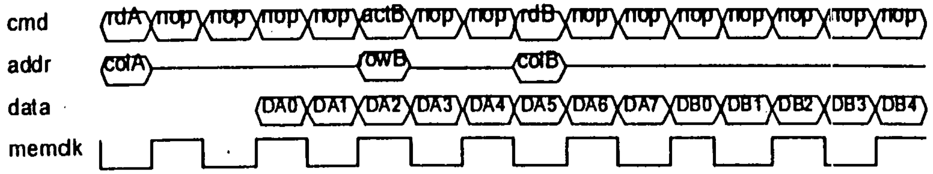
standard access operation for sequential read, using full page burst length

used for :
- display block access for block to line conversion in no compression mode, field picture - luma access with n = 16 - chroma access with n =8
- display block access for block to line conversion in 2M/3 mode - luma and chroma access with n = 16
- display block access for block to line conversion in M/2-H/2 mode, field picture - luma access with n = 4
- display read access for LMU - 32 pixels access with n = 16 - 16 pixels access with n = 8
- OSD read access with n = 32 (bitmap) or n = 8 (header)
- compressed data read for SCD with n = 16, for VLD with n = 32

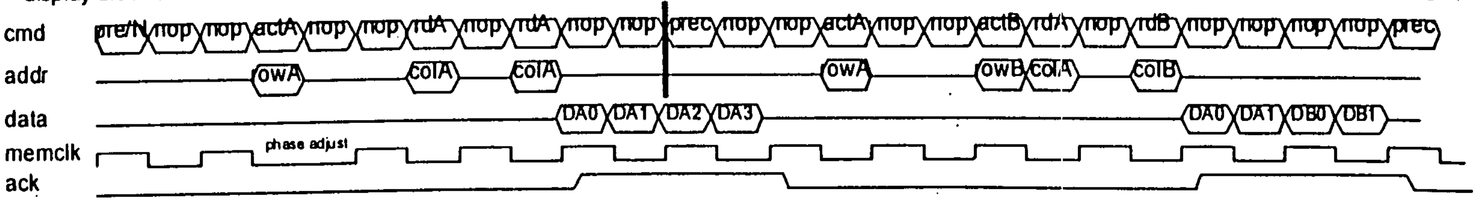
All display access for block to line conversion contain 2 read command of n words, in one or two banks.



If a bank access is needed (for display block access - field picture) this does not disturb the data flow



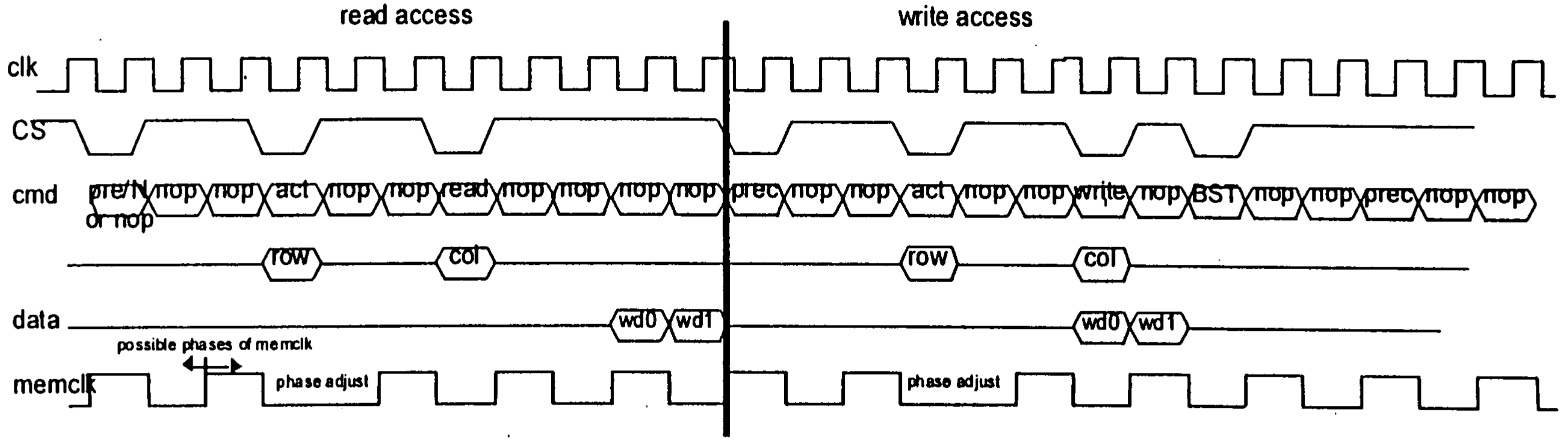
display block access for block to line conversion in H/2-M/2 mode - chroma access - without bank access inside (left) - with bank access inside (right)



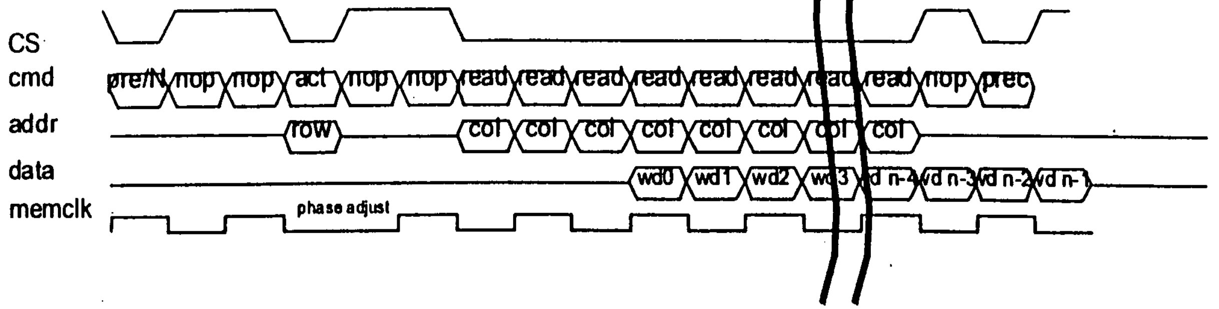
K3.2.2. Two Word Access -- read or write

2 word access, read or write

used for :
- local memory read (read)
- local memory write (write)
- standard definition input (write)
- block copy (read follow by write)



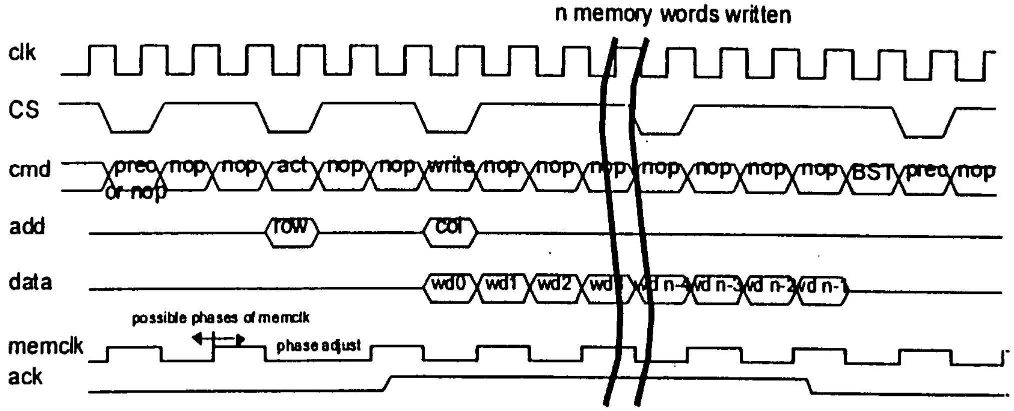
display block access for block to line conversion in no compression mode, progressive picture, luma n = 32, chroma n=8



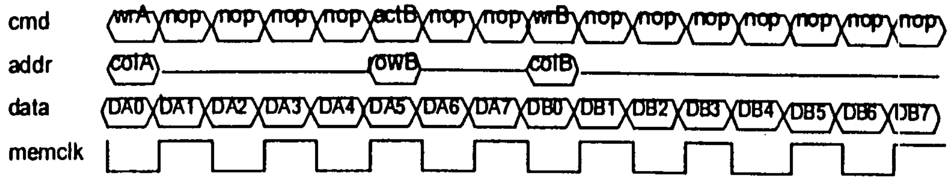
K3.2.3. Standard Access for Write

standard access operation for write, using burst length of 4

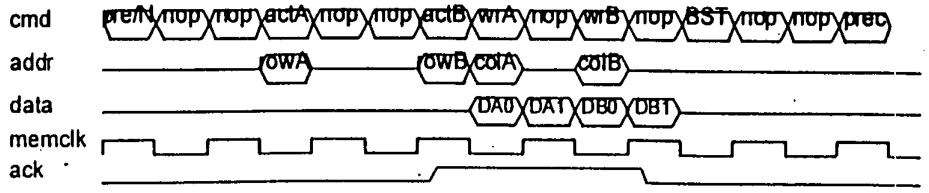
used for :
- reconstruction access in no compression mode, frame strucure pictures - luma access with n =32 - chrorna access with n = 16
- reconstruction access in 2M/3 mode - luma and chroma access with p = 8
- reconstruction access in M/2-H/2 mode - luma access with p = 2
- display write access from LMU - 32 pixels access with n = 16 - 16 pixels access with n = 8
- compressed data write with n = 16



If a bank access is needed (for MPEG2 field structure picture reconstruction) this does not disturb the data flow

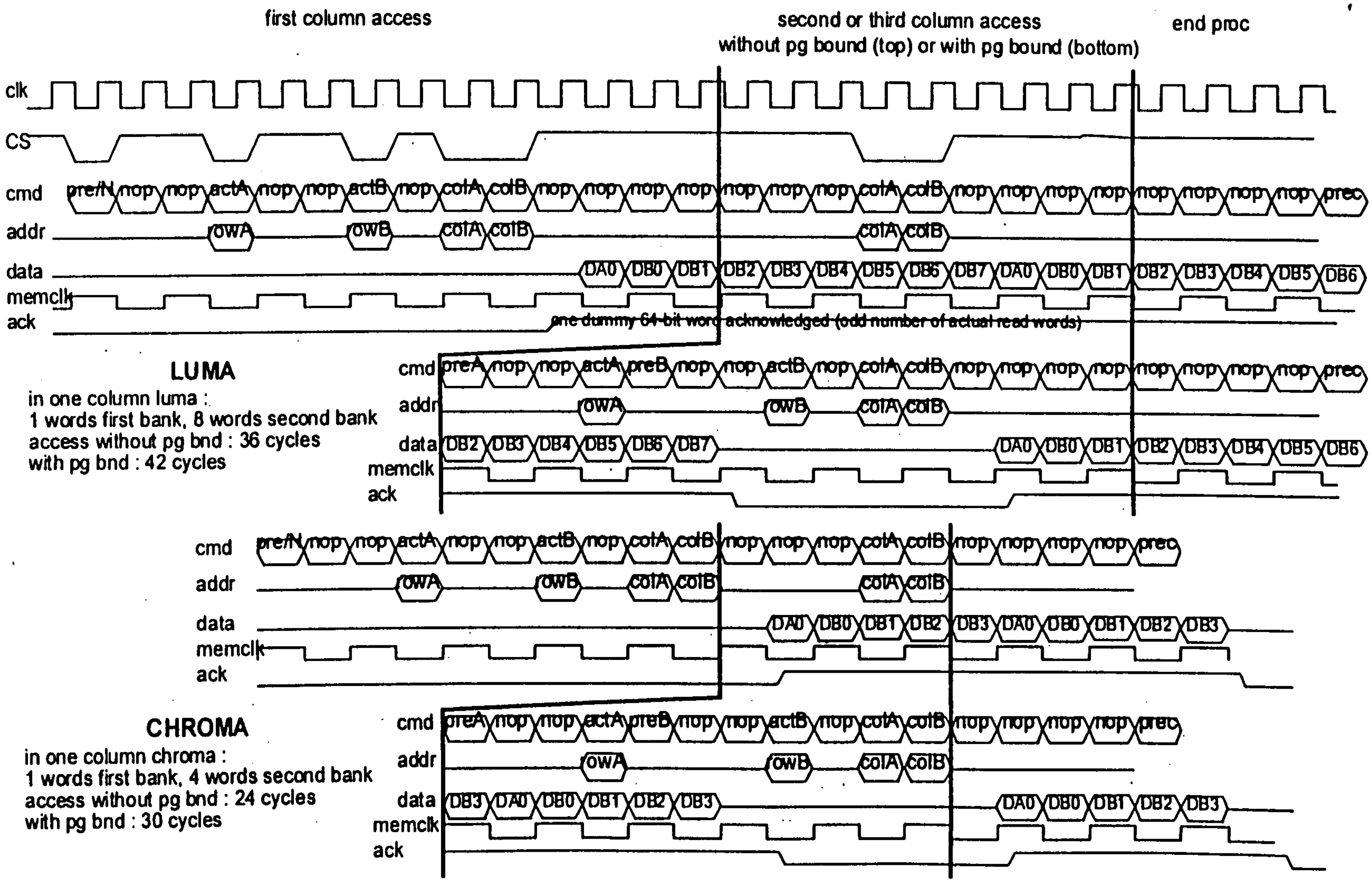


reconstruction of field structure picture in M/2-H/2 mode - chroma access - bank access inside burst



K3.2.4. prediction access in standard mode, column luma with 1 word first bank, 8 second bank, column chroma with 1 word first bank, 4 second bank

prediction access operation (3 column access) - LUMA : 1-8 - CHROMA : 1-4


first column access
second or third column access
without pg bound (top) or with pg bound (bottom)
end proc
clk
CS
cmd
addr
data
memclk
ack
pre/N
nop
actA
actB
colA
colB
preA
preB
prec
rowA
rowB
DA0
DB0
DB1
DB2
DB3
DB4
DB5
DB6
DB7
one dummy 64-bit word acknowledged (odd number of actual read words)

LUMA

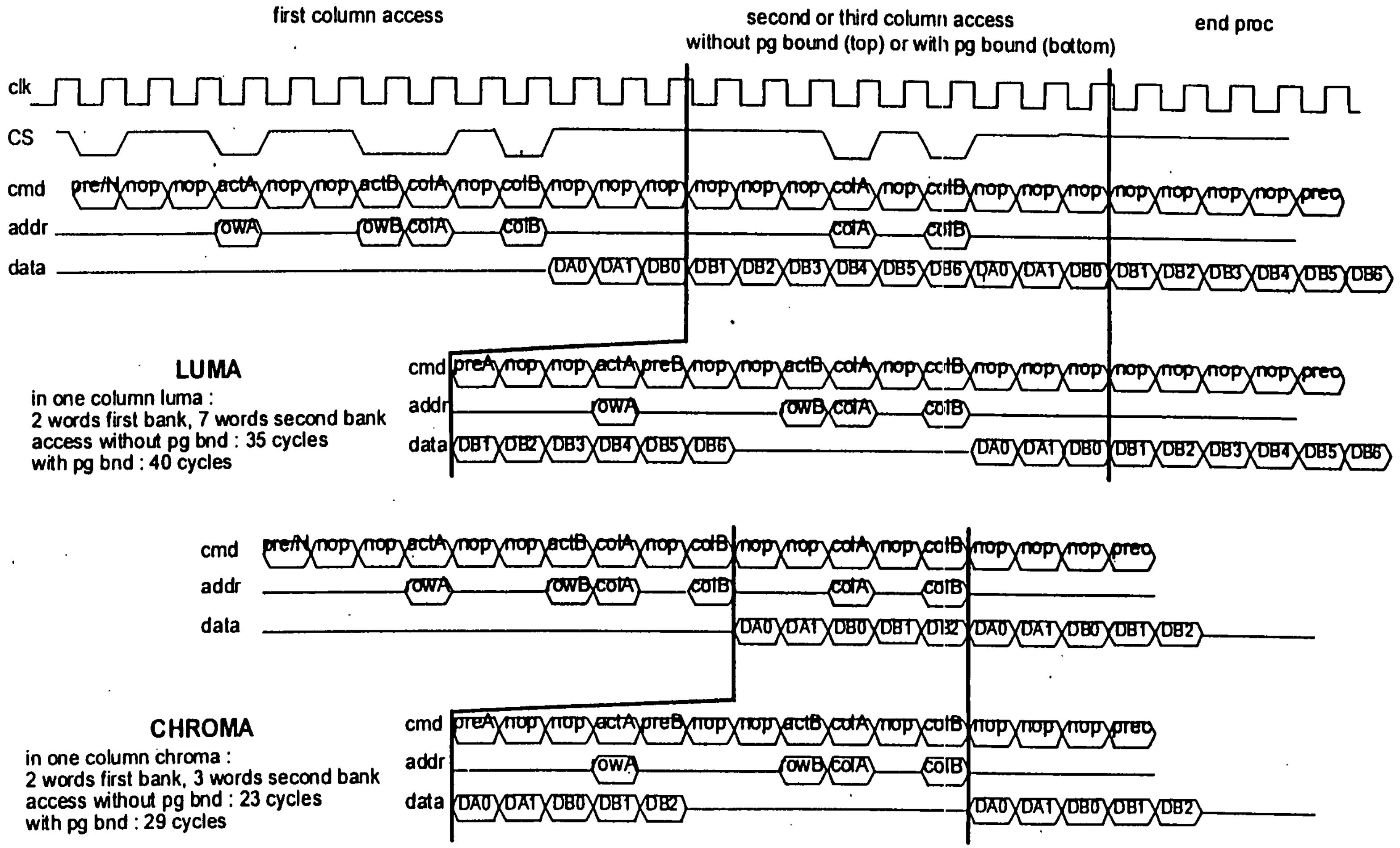
in one column luma :
1 words first bank, 8 words second bank
access without pg bnd : 36 cycles
with pg bnd : 42 cycles

CHROMA

in one column chroma :
1 words first bank, 4 words second bank
access without pg bnd : 24 cycles
with pg bnd : 30 cycles

K3.2.5. prediction access in standard mode, column luma with 2 word first bank, 7 second bank, column chroma with 2 word first bank, 3 second bank

prediction access operation (3 column access) - LUMA : 2-7 - CHROMA : 2-3



LUMA

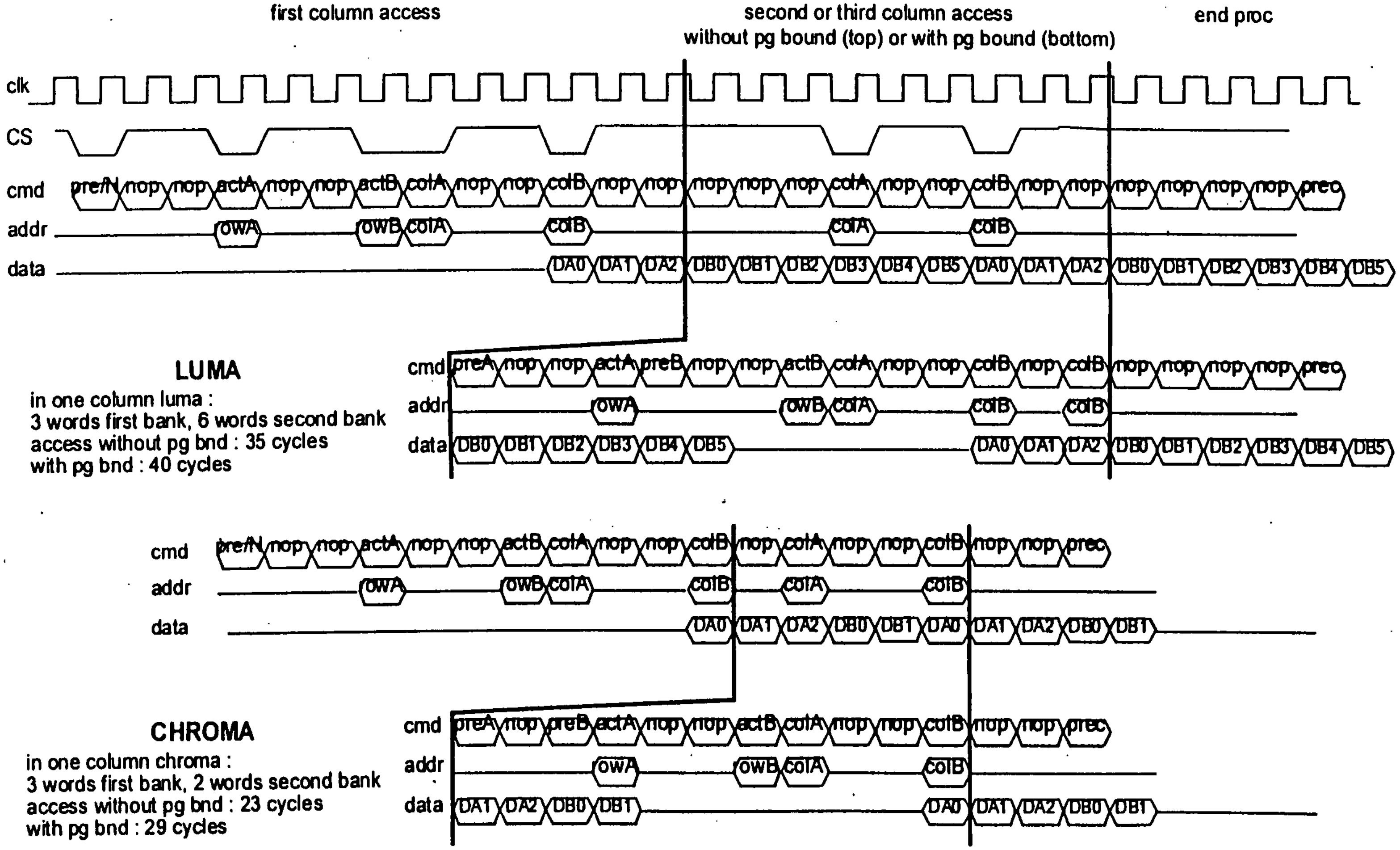
in one column luma :
2 words first bank, 7 words second bank
access without pg bnd : 35 cycles
with pg bnd : 40 cycles

CHROMA

in one column chroma :
2 words first bank, 3 words second bank
access without pg bnd : 23 cycles
with pg bnd : 29 cycles

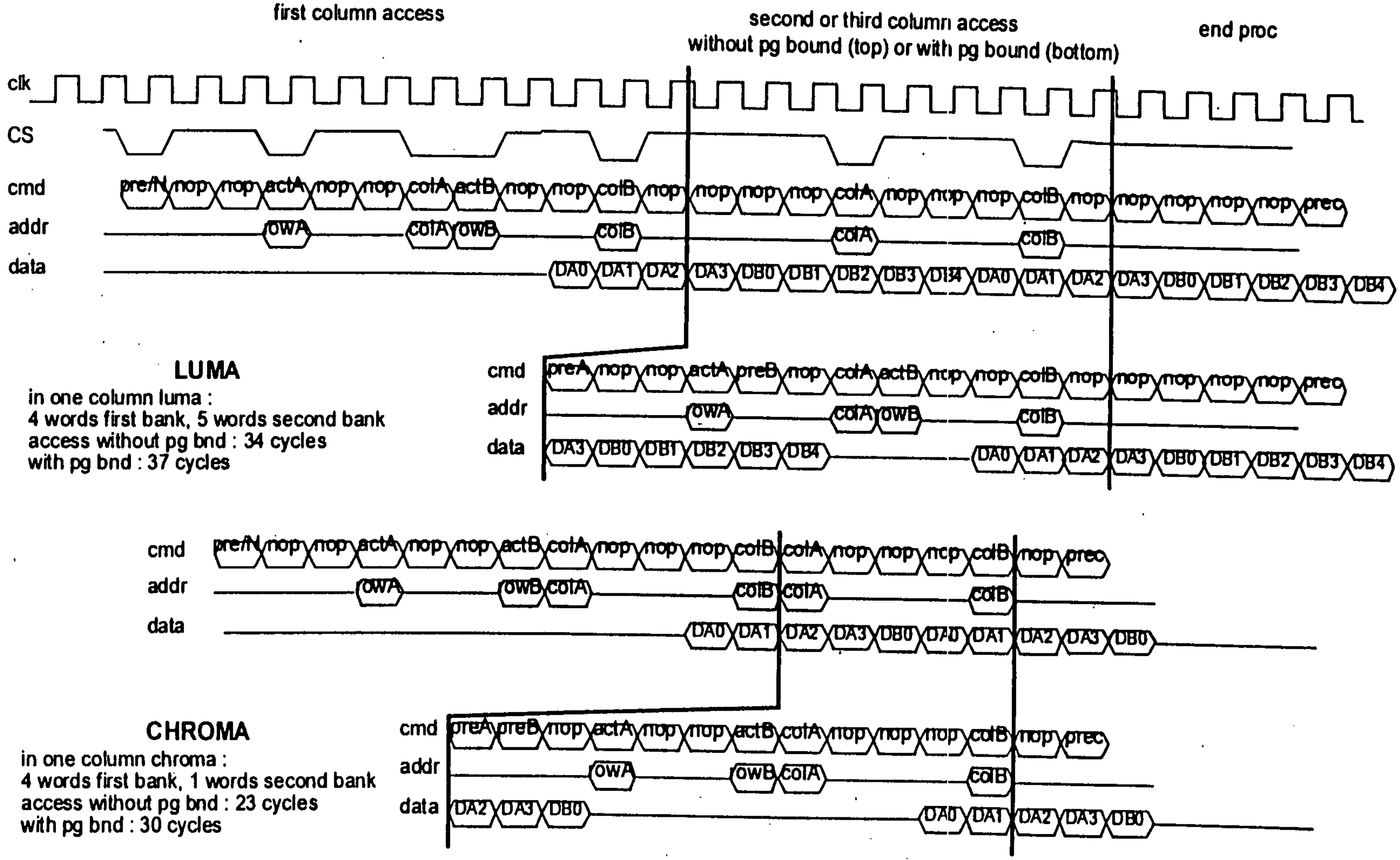
K3.2.6. prediction access in standard mode, column luma with 3 word first bank, 6 second bank, column chroma with 3 word first bank, 2 second bank

prediction access operation (3 column access) - LUMA : 3-6 - CHROMA : 3-2


first column access
second or third column access
without pg bound (top) or with pg bound (bottom)
end proc
clk
CS
cmd
addr
data
pre/N nop nop actA nop nop actB colA nop nop colB nop nop nop nop nop colA nop nop colB nop nop nop nop nop nop prec
owA
owB colA
colB
colA
colB
DA0 DA1 DA2 DB0 DB1 DB2 DB3 DB4 DB5 DA0 DA1 DA2 DB0 DB1 DB2 DB3 DB4 DB5
LUMA
in one column luma :
3 words first bank, 6 words second bank
access without pg bnd : 35 cycles
with pg bnd : 40 cycles
preA nop nop actA preB nop nop actB colA nop nop colB nop colB nop nop nop nop prec
DB0 DB1 DB2 DB3 DB4 DB5 DA0 DA1 DA2 DB0 DB1 DB2 DB3 DB4 DB5
pre/N nop nop actA nop nop actB colA nop nop colB nop colA nop nop colB nop nop prec
DA0 DA1 DA2 DB0 DB1 DA0 DA1 DA2 DB0 DB1
CHROMA
in one column chroma :
3 words first bank, 2 words second bank
access without pg bnd : 23 cycles
with pg bnd : 29 cycles
preA nop preB actA nop nop actB colA nop nop colB nop nop prec
DA1 DA2 DB0 DB1 DA0 DA1 DA2 DB0 DB1

K3.2.7. prediction access in standard mode, column luma with 4 word first bank, 5 second bank, column chroma with 4 word first bank, 1 second bank

prediction access operation (3 column access) - LUMA : 4-5 - CHROMA : 4-1


first column access
second or third column access
without pg bound (top) or with pg bound (bottom)
end proc
clk
CS
cmd
pre/N nop nop actA nop nop colA actB nop nop colB nop nop nop nop colA nop nop nop colB nop nop nop nop nop prec
addr
owA colA owB colB colA colB
data
DA0 DA1 DA2 DA3 DB0 DB1 DB2 DB3 DB4 DA0 DA1 DA2 DA3 DB0 DB1 DB2 DB3 DB4
cmd
preA nop nop actA preB nop colA actB nop nop colB nop nop nop nop nop prec
addr
owA colA owB colB
data
DA3 DB0 DB1 DB2 DB3 DB4 DA0 DA1 DA2 DA3 DB0 DB1 DB2 DB3 DB4
cmd
pre/N nop nop actA nop nop actB colA nop nop nop colB colA nop nop nop colB nop prec
addr
owA owB colA colB colA colB
data
DA0 DA1 DA2 DA3 DB0 DA0 DA1 DA2 DA3 DB0
cmd
preA preB nop actA nop nop actB colA nop nop nop colB nop prec
addr
owA owB colA colB
data
DA2 DA3 DB0 DA0 DA1 DA2 DA3 DB0

LUMA

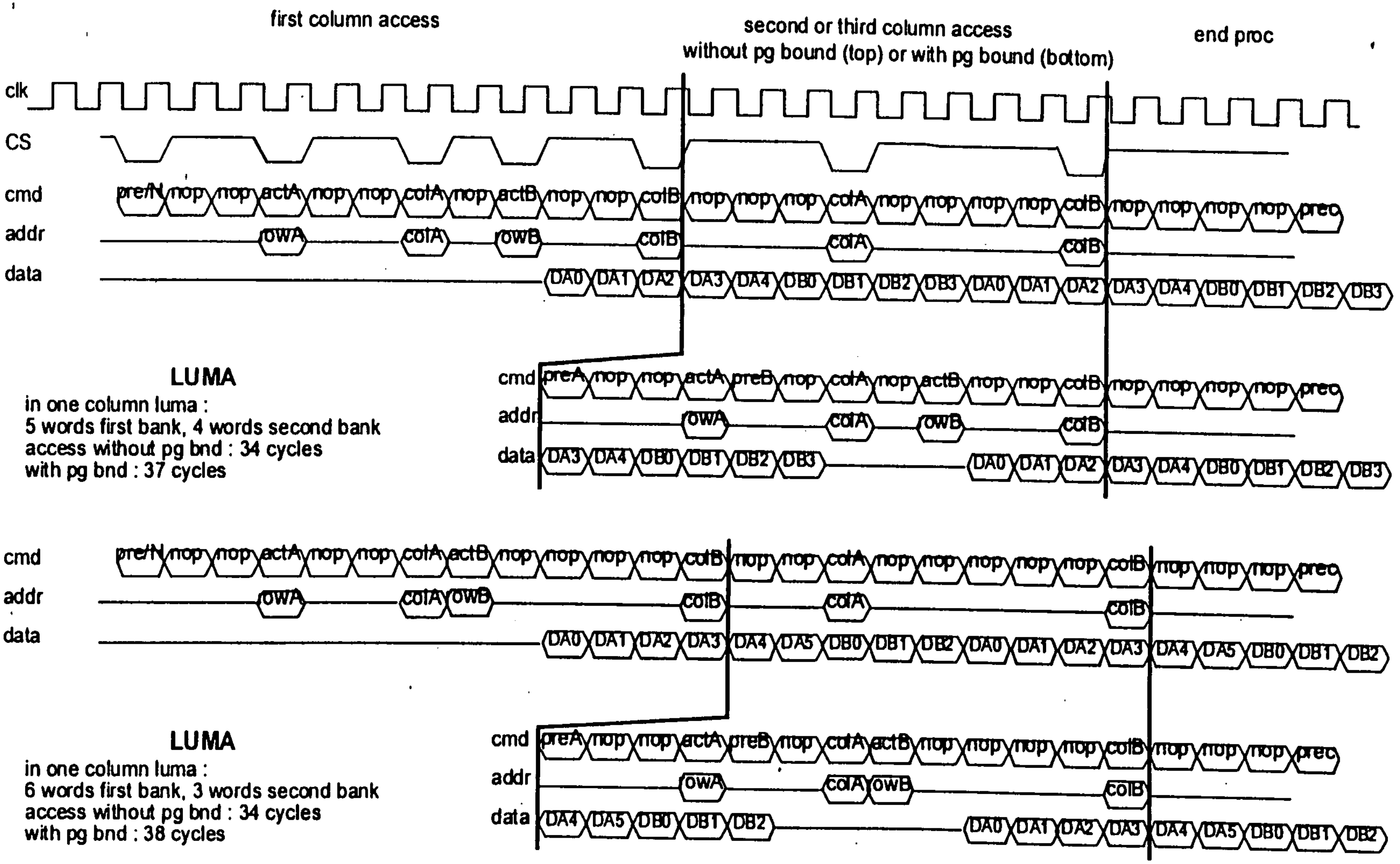
in one column luma :
4 words first bank, 5 words second bank
access without pg bnd : 34 cycles
with pg bnd : 37 cycles

CHROMA

in one column chroma :
4 words first bank, 1 words second bank
access without pg bnd : 23 cycles
with pg bnd : 30 cycles

K3.2.8. prediction access in standard mode, column luma with 5 word first bank, 4 second bank, and with 6 word first bank, 3 second bank

prediction access operation (3 column access) - LUMA : 5-4 - LUMA : 6-3


first column access
second or third column access
without pg bound (top) or with pg bound (bottom)
end proc
clk
CS
cmd
pre/N nop nop actA nop nop colA nop actB nop nop colB nop nop nop colA nop nop nop nop colB nop nop nop nop prec
addr
rowA colA rowB colB colA colB
data
DA0 DA1 DA2 DA3 DA4 DB0 DB1 DB2 DB3 DA0 DA1 DA2 DA3 DA4 DB0 DB1 DB2 DB3
cmd
preA nop nop actA preB nop colA nop actB nop nop colB nop nop nop nop prec
addr
rowA colA rowB colB
data
DA3 DA4 DB0 DB1 DB2 DB3 DA0 DA1 DA2 DA3 DA4 DB0 DB1 DB2 DB3
cmd
pre/N nop nop actA nop nop colA actB nop nop nop nop colB nop nop colA nop nop nop nop nop colB nop nop nop prec
addr
rowA colA rowB colB colA colB
data
DA0 DA1 DA2 DA3 DA4 DA5 DB0 DB1 DB2 DA0 DA1 DA2 DA3 DA4 DA5 DB0 DB1 DB2
cmd
preA nop nop actA preB nop colA actB nop nop nop nop colB nop nop nop prec
addr
rowA colA rowB colB
data
DA4 DA5 DB0 DB1 DB2 DA0 DA1 DA2 DA3 DA4 DA5 DB0 DB1 DB2

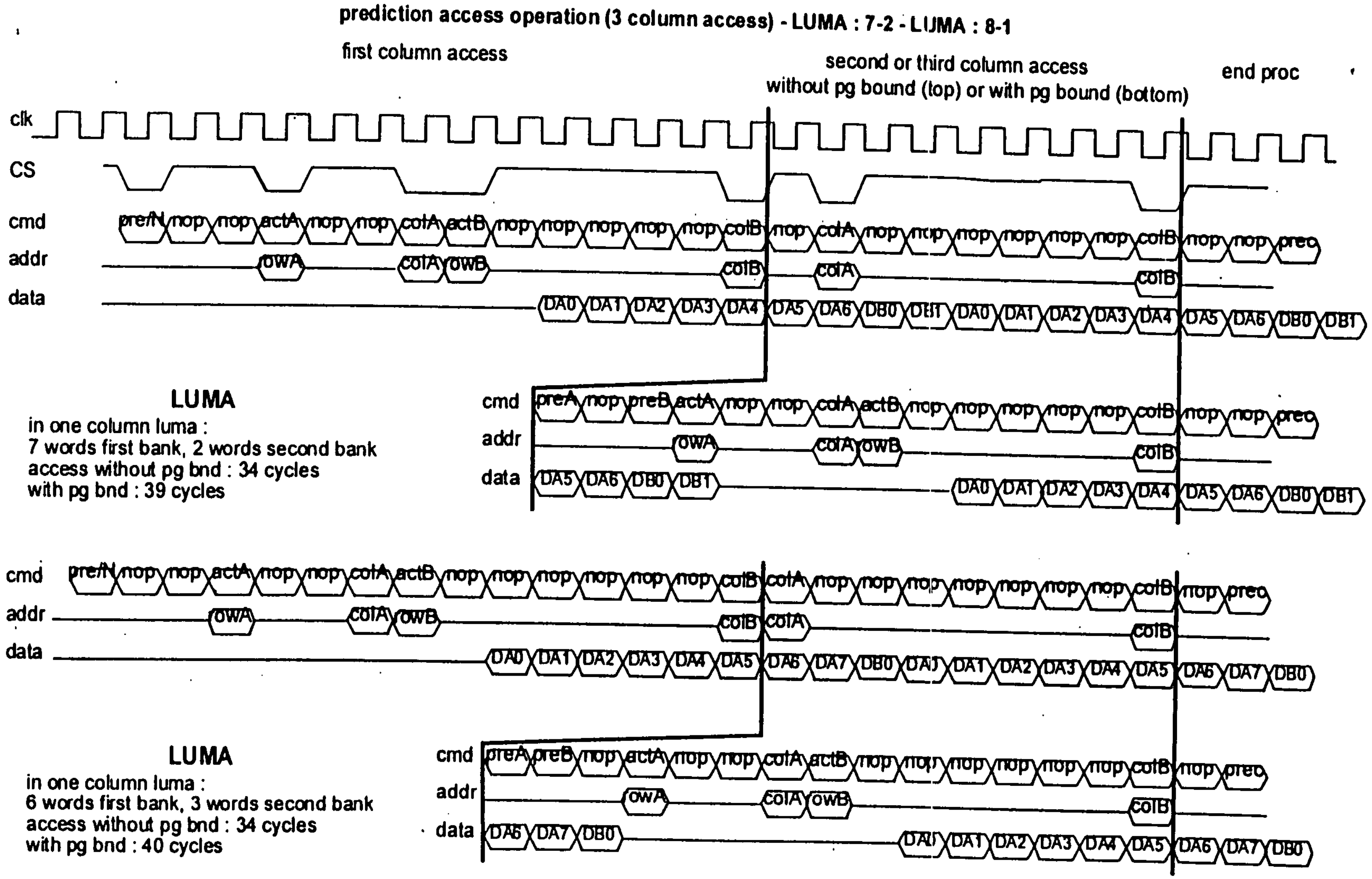
LUMA

in one column luma :
5 words first bank, 4 words second bank
access without pg bnd : 34 cycles
with pg bnd : 37 cycles

LUMA

in one column luma :
6 words first bank, 3 words second bank
access without pg bnd : 34 cycles
with pg bnd : 38 cycles

K3.2.9. prediction access in standard mode, column luma with 7 word first bank, 2 second bank, and with 8 word first bank, 1 second bank

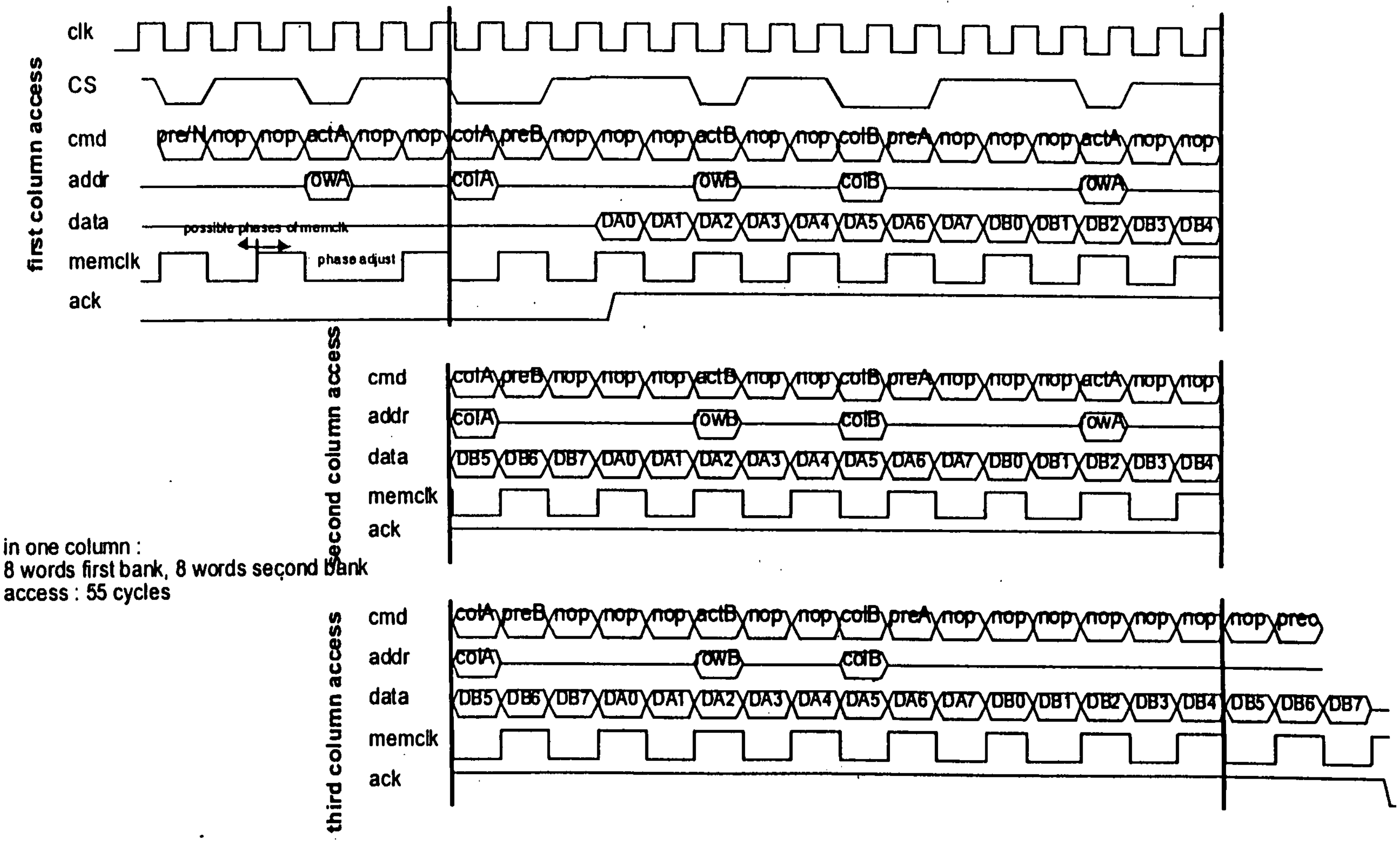

prediction access operation (3 column access) - LUMA : 7-2 - LUMA : 8-1
first column access
second or third column access
without pg bound (top) or with pg bound (bottom)
end proc
clk
CS
cmd
addr
data
pre/N nop nop actA nop nop colA actB nop nop nop nop nop colB nop colA nop nop nop nop nop nop colB nop nop prec
owA colA owB colB colA colB
DA0 DA1 DA2 DA3 DA4 DA5 DA6 DB0 DB1 DA0 DA1 DA2 DA3 DA4 DA5 DA6 DB0 DB1
preA nop preB actA nop nop colA actB nop nop nop nop nop colB nop nop prec
owA colA owB colB
DA5 DA6 DB0 DB1 DA0 DA1 DA2 DA3 DA4 DA5 DA6 DB0 DB1
pre/N nop nop actA nop nop colA actB nop nop nop nop nop nop colB colA nop nop nop nop nop nop nop colB nop prec
owA colA owB colB colA colB
DA0 DA1 DA2 DA3 DA4 DA5 DA6 DA7 DB0 DA0 DA1 DA2 DA3 DA4 DA5 DA6 DA7 DB0
preA preB nop actA nop nop colA actB nop nop nop nop nop nop colB nop prec
owA colA owB colB
DA6 DA7 DB0 DA0 DA1 DA2 DA3 DA4 DA5 DA6 DA7 DB0

LUMA
in one column luma :
7 words first bank, 2 words second bank
access without pg bnd : 34 cycles
with pg bnd : 39 cycles

LUMA
in one column luma :
6 words first bank, 3 words second bank
access without pg bnd : 34 cycles
with pg bnd : 40 cycles

K3.2.10. prediction access in 2M/3 mode

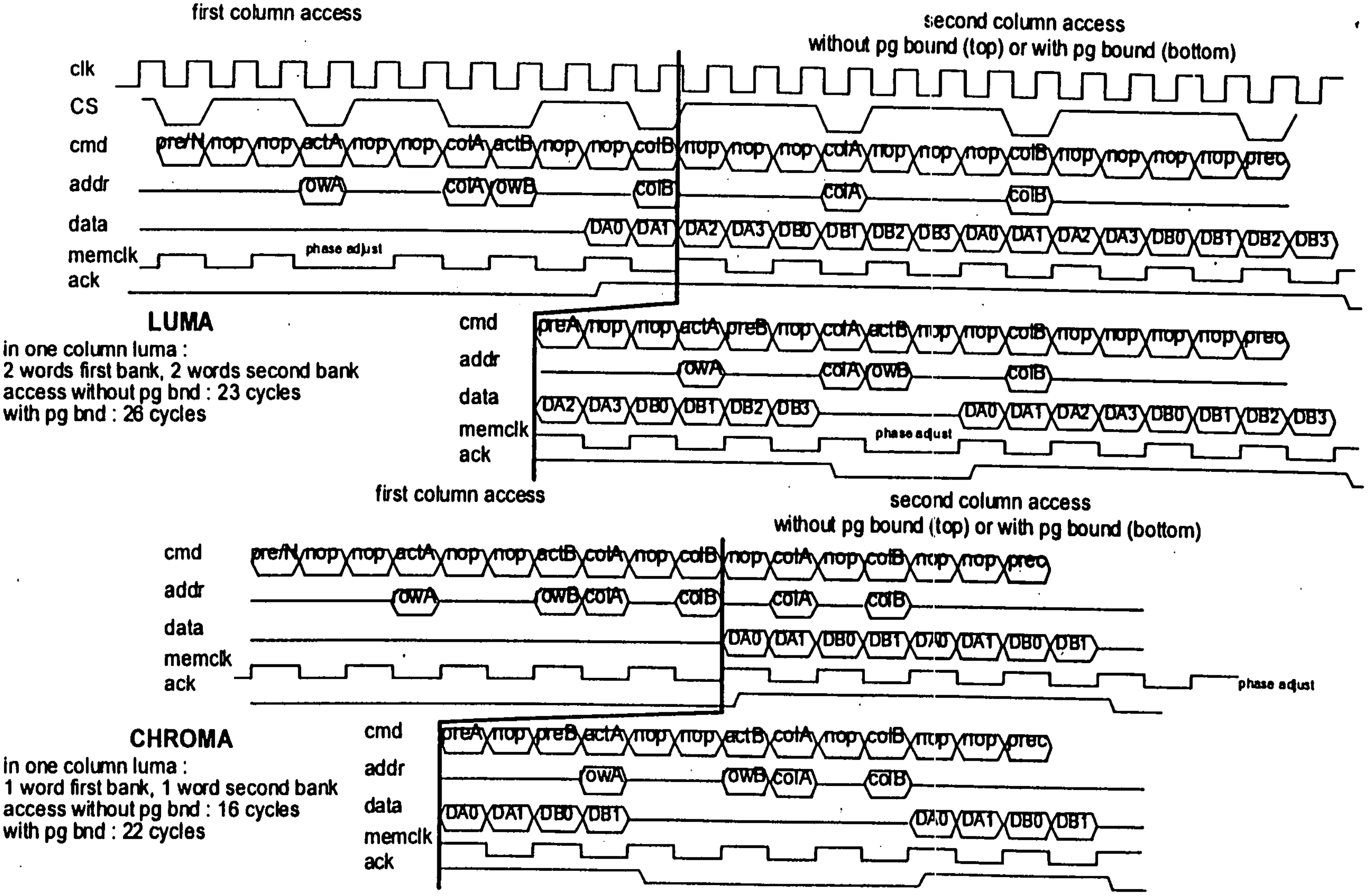
prediction access operation (3 column access) - 2M/3 MODE (33% compression)


first column access
clk
CS
cmd
pre/N nop nop actA nop nop colA preB nop nop nop actB nop nop colB preA nop nop nop actA nop nop
addr
owA colA owB colB owA
data
possible phases of memclk
DA0 DA1 DA2 DA3 DA4 DA5 DA6 DA7 DB0 DB1 DB2 DB3 DB4
memclk
phase adjust
ack
second column access
cmd
colA preB nop nop nop actB nop nop colB preA nop nop nop actA nop nop
addr
colA owB colB owA
data
DB5 DB6 DB7 DA0 DA1 DA2 DA3 DA4 DA5 DA6 DA7 DB0 DB1 DB2 DB3 DB4
memclk
ack
third column access
cmd
colA preB nop nop nop actB nop nop colB preA nop nop nop nop nop nop nop preo
addr
colA owB colB
data
DB5 DB6 DB7 DA0 DA1 DA2 DA3 DA4 DA5 DA6 DA7 DB0 DB1 DB2 DB3 DB4 DB5 DB6 DB7
memclk
ack

in one column :
8 words first bank, 8 words second bank
access : 55 cycles

K3.2.11. prediction access in H/2-M/2 mode

prediction access operation (2 column access) - H/2-M/2 MODE (75% compression)

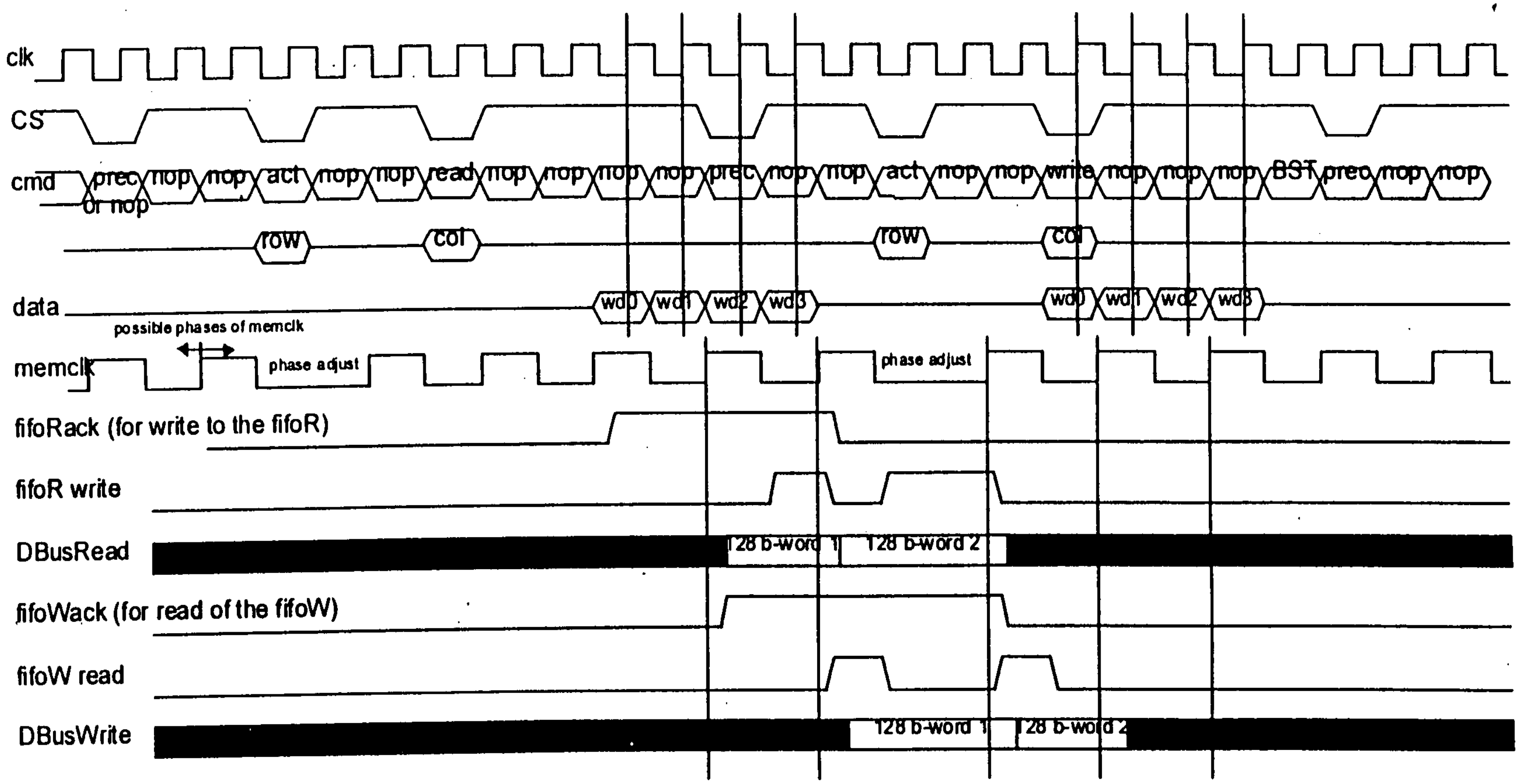

first column access
second column access
without pg bound (top) or with pg bound (bottom)
clk
CS
cmd
pre/N nop nop actA nop nop colA actB nop nop colB nop nop nop colA nop nop nop colB nop nop nop nop prec
addr
rowA colA rowB colB colA colB
data
DA0 DA1 DA2 DA3 DB0 DB1 DB2 DB3 DA0 DA1 DA2 DA3 DB0 DB1 DB2 DB3
memclk
phase adjust
ack
cmd
preA nop nop actA preB nop colA actB nop nop colB nop nop nop nop prec
addr
rowA colA rowB colB
data
DA2 DA3 DB0 DB1 DB2 DB3 DA0 DA1 DA2 DA3 DB0 DB1 DB2 DB3
memclk
phase adjust
ack
first column access
second column access
without pg bound (top) or with pg bound (bottom)
cmd
pre/N nop nop actA nop nop actB colA nop colB nop colA nop colB nop nop prec
addr
rowA rowB colA colB colA colB
data
DA0 DA1 DB0 DB1 DA0 DA1 DB0 DB1
memclk
ack
phase adjust
cmd
preA nop preB actA nop nop actB colA nop colB nop nop prec
addr
rowA rowB colA colB
data
DA0 DA1 DB0 DB1 DA0 DA1 DB0 DB1
memclk
ack

LUMA
in one column luma :
2 words first bank, 2 words second bank
access without pg bnd : 23 cycles
with pg bnd : 26 cycles

CHROMA
in one column luma :
1 word first bank, 1 word second bank
access without pg bnd : 16 cycles
with pg bnd : 22 cycles

K3.2.12. Example diagram write following read operation

Example write following read operation



HD MPEG VIDEO DECODER

APPENDIX L

MEMORY MANAGEMENT

Thomson Consumer Electronics, Inc.
Indianapolis, Indiana, USA

Revision No. 2.1

1. Introduction

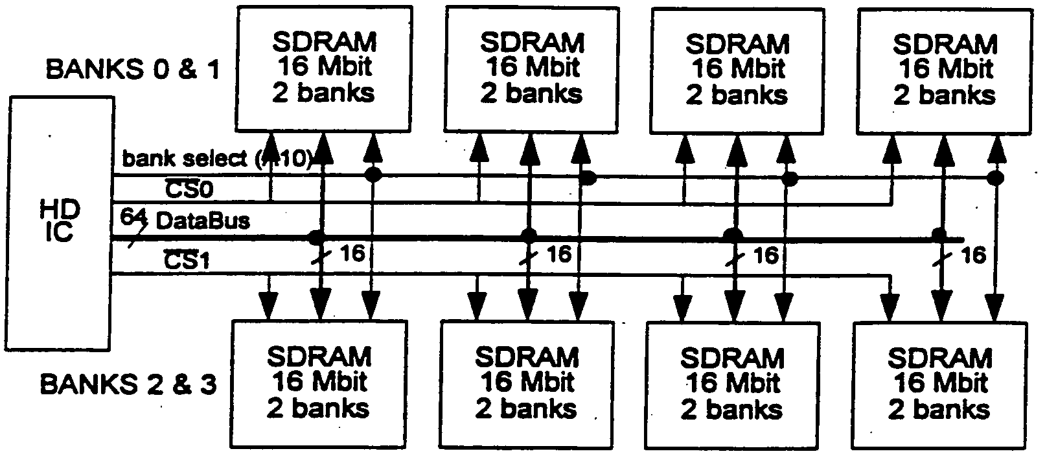
This document describes the memory management of the HD MPEG IC. Memory bus is 64 bit wide, memory used is of synchronous DRAM type. The generic use is a latency of 3 and a burst length of 4.

2. General mapping examples

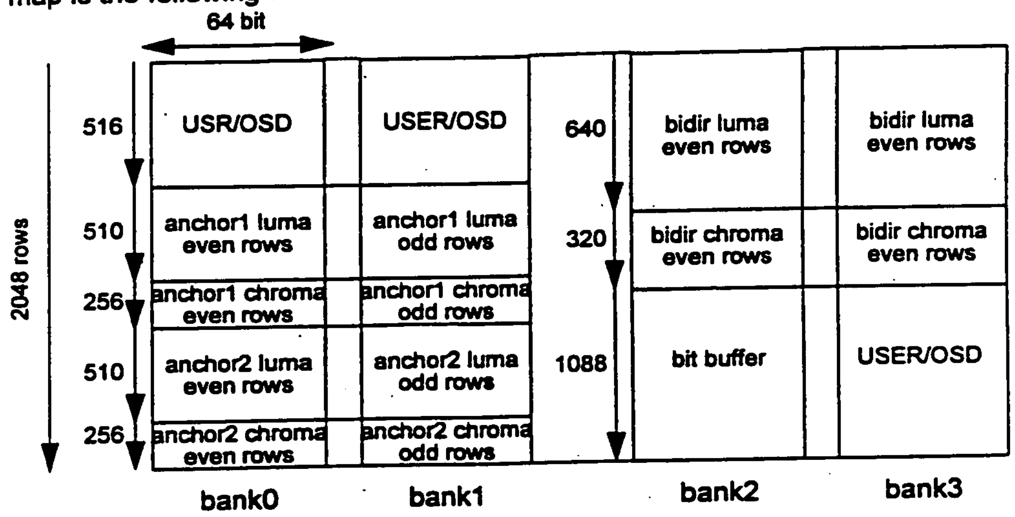
The purpose of this paragraph is to give memory mapping example for the most constraining application (i.e. 1920x1088 full HD picture decoding) in the different memory modes

2.1. 128 Mbit of memory - No compression mode

The memory is organised in two 64-Mbit banks, each one using four 16-Mbit SDRAM. Each SDRAM contains two banks that can be used to hide page accesses.

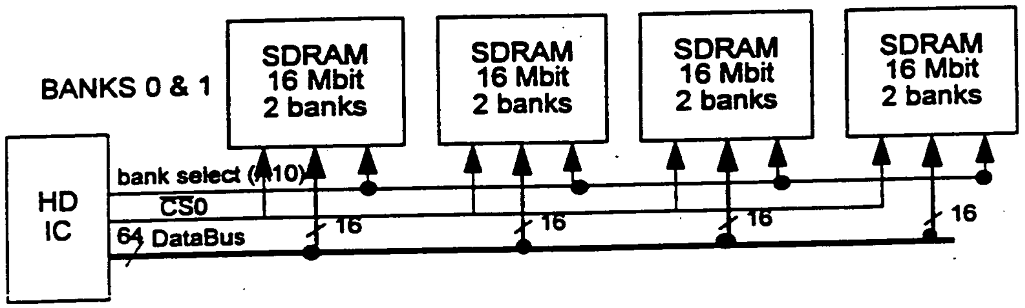


The logical map is the following :

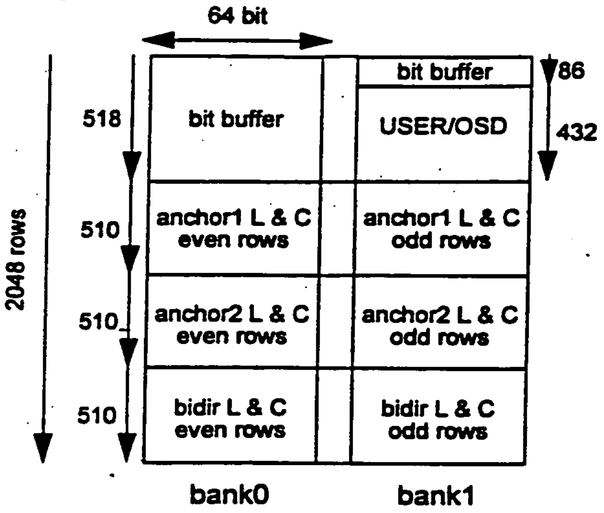


2.2. *64 Mbit of memory - 2M/3 mode (33 % compression)*

The memory is organised in one 64-Mbit banks, using four 16-Mbit SDRAM.



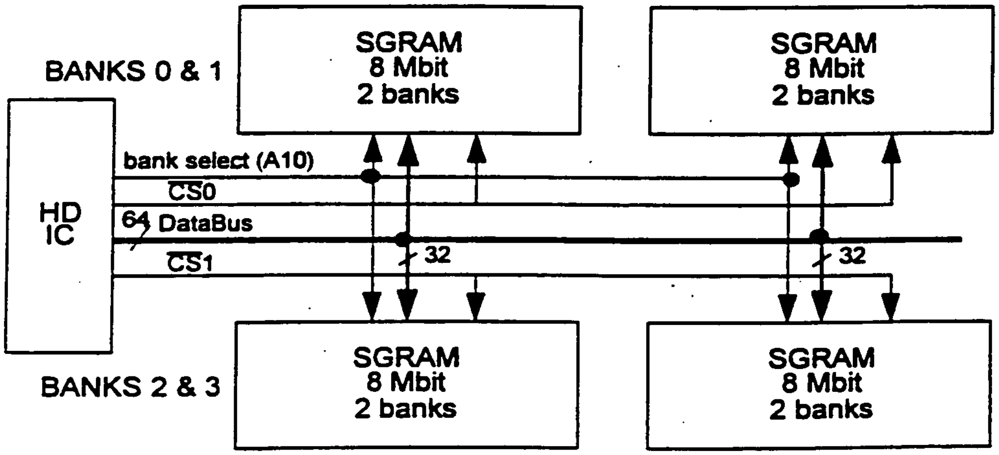
The logical map is the following :



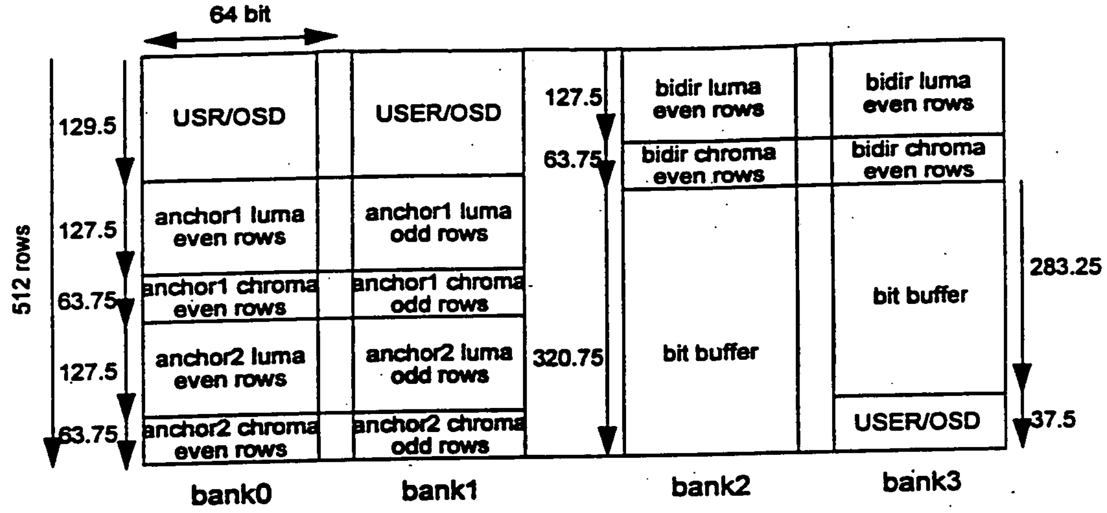
The bit buffer is split in two parts to meet the bit buffer size requirement.

2.3. 32 Mbit of memory - M/2-H/2 mode (75 % compression)

The memory is organised in two 16-Mbit banks, each one using two 8-Mbit SGRAM. Each SGRAM contains two banks that can be used to hide page accesses.



The logical map is the following :

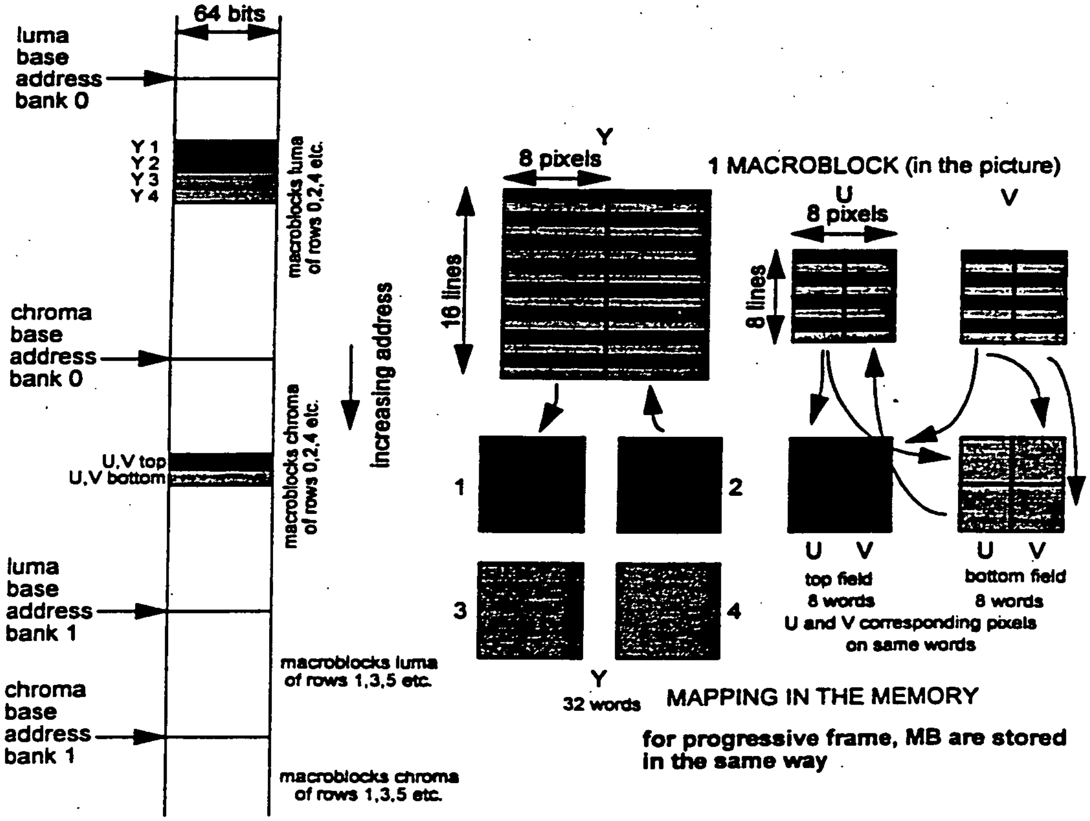


The bit buffer is split in two parts to meet the bit buffer size requirement.

3. *Memory organization for picture*

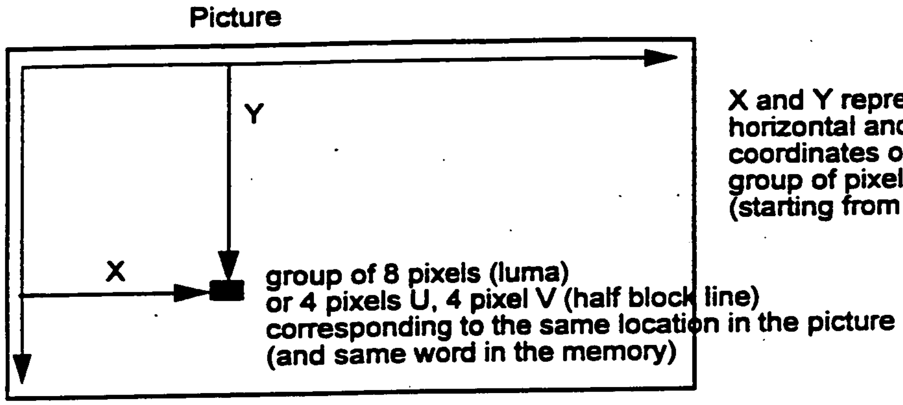
3.1. Normal mode (No compression)

Inside the macroblock entity, datas are grouped by field. Macroblocks are grouped by row in the picture. Every other row is stored in bank 0 of SDRAM, alternating with the others in bank 1.


64 bits
luma base address bank 0
Y 1
Y 2
Y 3
Y 4
macroblocks luma of rows 0,2,4 etc.
chroma base address bank 0
U,V top
U,V bottom
macroblocks chroma of rows 0,2,4 etc.
increasing address
luma base address bank 1
macroblocks luma of rows 1,3,5 etc.
chroma base address bank 1
macroblocks chroma of rows 1,3,5 etc.
Y
8 pixels
16 lines
1 MACROBLOCK (in the picture)
U
8 pixels
V
8 lines
1
2
3
4
Y
32 words
U V
top field
8 words
U V
bottom field
8 words
U and V corresponding pixels on same words
MAPPING IN THE MEMORY

for progressive frame, MB are stored in the same way

Address calculation


Picture
Y
X
group of 8 pixels (luma)
or 4 pixels U, 4 pixel V (half block line)
corresponding to the same location in the picture
(and same word in the memory)

X and Y represent the horizontal and vertical coordinates of the group of pixels (starting from (0,0)).

A segment is an amount of memory of 256 bytes, or 32 64-bits words. It is equivalent to the amount of memory to store a macroblock of luminance or two of chrominance.
W = width of the picture in macroblocks
FP = frame pointer = start address (in segment unit) of the considered frame buffer (the luma base address).
FS = frame size in macroblocks
FA = frame pointer adjustment, is a programmable value used to shift chroma base address.
field = 0 for top field, 1 for bottom field
bank = 0 for bank A, 1 for bank B of the SDRAM devices. bank select the specified bank in the device.

Following operations are integer operations.

The base address of both banks are the same.

LUMA GROUP OF 8 PIXELS :

bank = (Y/16) mod 2
segment = (Y/32) x W + (X/2) + FP
field = Y mod 2

In the specified bank, the address is :

address = segmentx32 + fiel x16 + (X mod 2)x8 + (Y mod 16) / 2

CHROMA GROUP OF 8 PIXELS (4U, 4V) :

bank = (Y/8) mod 2
segment = ((Y/16) x W + X/2) / 2 + FP + (FS+1)/2 + FA
useg = ((Y/16) x W + X/2) mod 2
field = Y mod 2

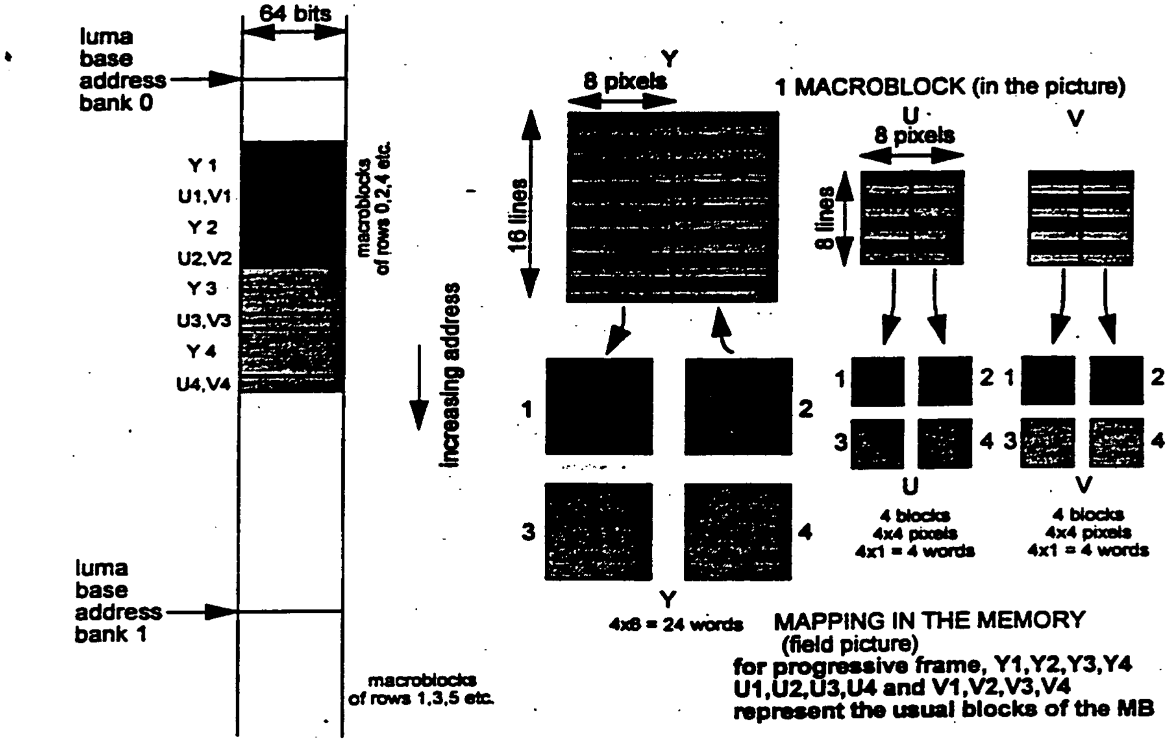
In the specified bank, the address is :

address = segment x 32 + useg x 16 + field x 8 + (X mod 2) x 4 + (Y mod 8) / 2

With 12 row address bit and 8 column address bit SDRAM, 8 segments can be stored in one row, i.e. the luma of 8 macroblocks and the chroma of 16.

3.2. Mode 2M/3 (33% compression)

The global scheme is kept as in normal mode, except the fact that luma and chroma are stored in the same place for a macroblock. In this case, the luma data represents 75 % of the luma data of a non compressed buffer, i.e. 24 memory words per macroblock. The chroma data is 50 % of that of a non compressed buffer, i.e. 8 words per macroblock. The resulting luma + chroma macroblock is 32 words, i.e. one segment.



Address calculation :

Using the same notations as above :

LUMA PIXELS :

bank = (Y/16) mod 2
segment = (Y/32) x W + (X/2) + FP
field = Y mod 2

In the specified bank, the start addresses of the 6-words compressed blocks are :

for field pictures : address = segment x 32 + field x 16 + (X mod 2) x 8
for frame picture : address = segment x 32 + ((Y/8) mod 2) x 16 + (X mod 2) x 8

CHROMA PIXELS :

bank = (Y/8) mod 2
segment = (Y/16) x W + X/2 + FP
field = Y mod 2

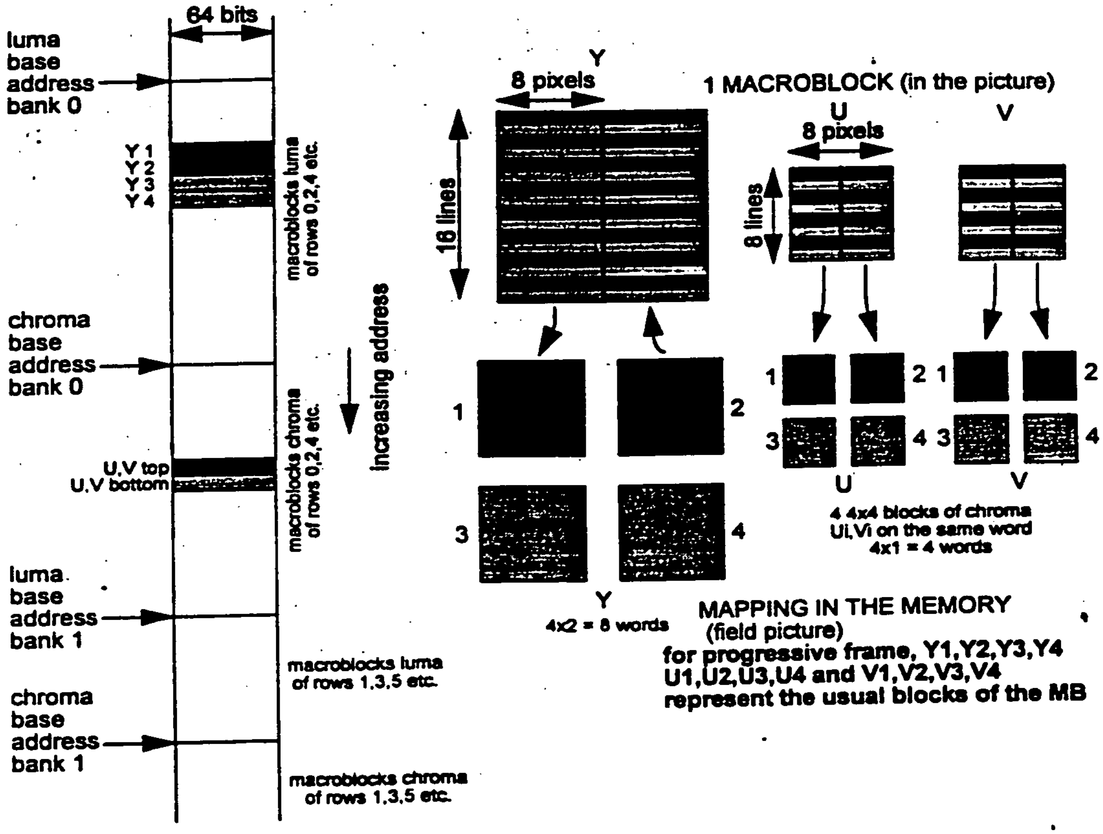
V = 1 for chroma V, 0 for chroma U

In the specified bank, the addresses of the 1-word compressed blocks are :

for field pictures : address = segment x 32 + field x 16 + (X mod 2) x 8 + 6 + V
for frame pictures : address = segment x 32 + ((Y/4) mod 2)x16 + (X mod 2)x8 + 6 + V

3.3. *Mode M/2, H/2 (75% compression)*

The global scheme is kept as in normal mode. In this case, the luma data represents 25 % of the luma data of a non compressed buffer, i.e. 8 memory words per macroblock. The chroma data is 25 % of that of a non compressed buffer, i.e. 4 words per macroblock. One chroma 4x4 block represents 32 bits. The U and V blocks on the same place are concatenated in the same memory word (U = MSB, V = LSB).


64 bits
luma base address bank 0
Y 1
Y 2
Y 3
Y 4
macroblocks luma of rows 0,2,4 etc.
chroma base address bank 0
U,V top
U,V bottom
macroblocks chroma of rows 0,2,4 etc.
increasing address
luma base address bank 1
macroblocks luma of rows 1,3,5 etc.
chroma base address bank 1
macroblocks chroma of rows 1,3,5 etc.
Y
8 pixels
16 lines
1 MACROBLOCK (in the picture)
U
8 pixels
8 lines
V
1
2
3
4
1
2
3
4
1
2
3
4
U
V
4 4x4 blocks of chroma
Ui,Vi on the same word
4x1 = 4 words
Y
4x2 = 8 words

MAPPING IN THE MEMORY
(field picture)
for progressive frame, Y1,Y2,Y3,Y4
U1,U2,U3,U4 and V1,V2,V3,V4
represent the usual blocks of the MB

Address calculation :

Using the same notations as above :

LUMA PIXELS :

bank = (Y/16) mod 2

$segment = ((Y/32) \times W + (X/2)) / 4 + FP$
$useg = ((Y/32) \times W + (X/2)) \bmod 4$
$field = Y \bmod 2$

In the specified bank, the start addresses of the 2-words compressed blocks are :

field pictures : $address = segment \times 32 + useg \times 8 + field \times 4 + (X \bmod 2) \times 2$
frame picture : $address = segment \times 32 + useg \times 8 + ((Y/8) \bmod 2) \times 4 + (X \bmod 2) \times 2$

CHROMA PIXELS :

$bank = (Y/8) \bmod 2$
$segment = (((Y/16) \times W + X/2)/ 2+ (FS+1)/2)/ 4 + FP + FA$
$useg = ((Y/16) \times W + X/2 + ((FS+1)/2 \bmod 4) \times 2) \bmod 8$
$field = Y \bmod 2$

In the specified bank, the addresses of the 1-word 2-compressed blocks(U,V) are :

field pictures : $address = segment \times 32 + useg \times 4 + field \times 2 + X \bmod 2$
frame picture : $address = segment \times 32 + useg \times 4 + ((Y/4) \bmod 2) \times 2 + X \bmod 2$

Note :

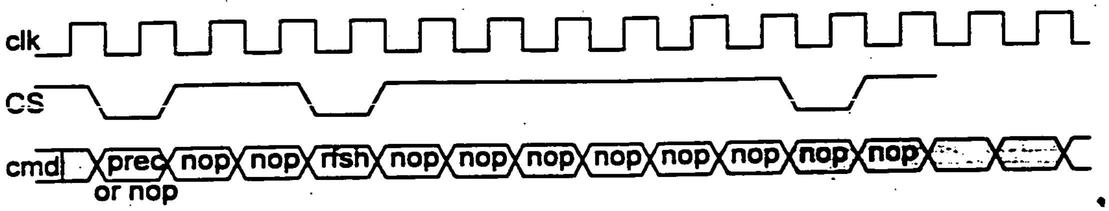
The field or frame picture flag is set by a register.

4. Memory Access Description

SDRAM is used in configuration : latency = 3, burst length = 4.

4.1. Refresh

10 cycles



4.2. Memory interface read/write

- read access : 128 bits (2 words) : 10 cycles
- write access : 128 bits (2 words) : 10 cycles

4.3. Standard definition video access

- luma access : 16 pixels - 1 line of a MB (2words) : 10 cycles
- chroma access : 8 pixels U, 8 pixels V - 1 line of a MB (2words) : 10 cycles

4.4. Display access

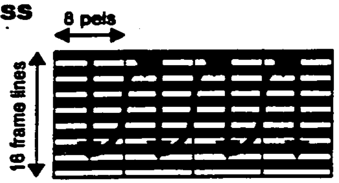
4.4.a. Block access for block to line conversion

No compression mode

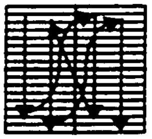
Standard read access with burst length 4 is used. FP is set even to avoid page access during 2 macroblocks read.

- 4 luma block access (2 MB field for field pictures, 1 MB for progr frames - 32 words) : 39 cycles

pixel access



2 MB field
(field picture - field display)



1 MB frame
(progressive frame - progr display)
(one "field" after the other)

Addressing :

Rn_j = row number of the accessed macroblock (vertical position of the accessed MB)
MBR_j = horizontal position of the accessed MB
bank = Rn_j mod 2
field = specified by the display

Addresses sequence (global address in specified bank from which row and column addresses are extracted) :

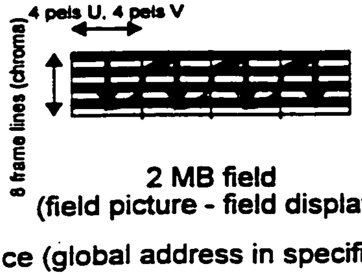
for field picture :
for j := 0 upto 1 { for i := 0 upto 3 {
add = (Rn_j/2 x W + MBR_j + FP) x 32 + field x 16 + i x 4 } }

There may be a bank access inside, without effect on the length of the process.

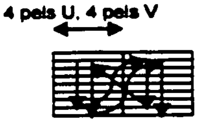
for progressive picture :
for k := 0 upto 1 { for i := 0 upto 3 {
add = (Rn/2 x W + MBR + FP) x 32 + k x 16 + i x 4 } }

- chroma block access (2 MB field for field pictures, 1 MB for progr frames - 16 words) : 23 cycles

pixel access



2 MB field
(field picture - field display)



1 MB frame
(progressive frame - progr display)

Addresses sequence (global address in specified bank from which row and column addresses are extracted) :

for field picture :

```
for j := 0 upto 1 { for i := 0 upto 1 {
    add = $(FP + (FS+1)/2 + FA) \times 32 + (Rn_j/2 \times W + MBR_j) \times 16 + field \times 8 + i \times 4$ } }
```

There may be a bank access inside, without effect on the length of the process.

for progressive picture :

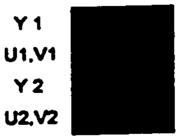
```
for k := 0 upto 1 { for i := 0 upto 1 {
    add = $(FP + (FS+1)/2 + FA) \times 32 + (Rn/2 \times W + MBR) \times 16 + k \times 8 + i \times 4$ } }
```

Mode 2M/3 (33% compression)

FP is set even to avoid page access during 2 macroblocks read.

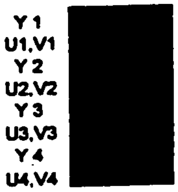
- 4 blocks access - luma and chroma (2 MB field for field pictures, 1 MB for progr frames - 32 words) : 39 cycles

pixel access



for 2 consecutives MB

2 MB field
(field picture - field display)



1 MB frame
(progressive frame - progr display)

Addresses sequence (global address in specified bank from which row and column addresses are extracted) :

bank = Rn_j mod 2

for field picture :

```
for j := 0 upto 1 { for i := 0 upto 3 {
    add = $(Rn_j/2 \times W + MBR_j + FP) \times 32 + field \times 16 + i \times 4$ } }
```

There may be a bank access inside, without effect on the length of the process.

for progressive picture :

```
for k := 0 upto 1 { for i := 0 upto 3 {
    add = $(Rn/2 \times W + MBR + FP) \times 32 + k \times 16 + i \times 4$ } }
```

Mode M/2, H/2 (75% compression)

Same accesses than no-compression mode concerning the blocks.

- 4 blocks access - luma and chroma (2 MB field for field pictures, 1 MB for progr frames - 8 words) : 15 cycles

Addresses sequence :

bank = Rn_j mod 2

for field picture :
for j := 0 upto 1 { add = $FP \times 32 + (Rn_j/2 \times W + MBR_j) \times 8 + field \times 4$ }

There may be a bank access inside, without effect on the length of the process.

for progressive picture :
for k := 0 upto 1 { add = $FP \times 32 + (Rn/2 \times W + MBR) \times 8 + k \times 4$ }

- chroma block access (2 MB field for field pictures, 1 MB for progr frames - 4 words) : 11 cycles if no bank access inside, 14 cycles if bank access (bank access can be avoided if the number of macroblocks per row of the picture is even)

Addresses sequence :

for field picture :
for j := 0 upto 1 { add = $(FP+FA)\times 32 + ((FS+1)/2\times 2 + Rn_j/2 \times W + MBR_j) \times 4 + field \times 2$}

There may be a bank access inside, without effect on the length of the process.

for progressive picture :
for k := 0 upto 1 { add = $(FP + FA) \times 32 + ((FS+1)/2\times 2 + Rn/2 \times W + MBR) \times 4 + k \times 2$}

4.4.b. Dislay access for LMU

- Write data from LMU :

32 pixel access : standard write access with incrementing address, (16 words) : 23 cycles
16 pixel access : standard write access with incrementing address, (8 words) : 15 cycles

- Read data for LMU :

32 pixel access : standard read access with incrementing address, (16 words) : 23 cycles
16 pixel access : standard read access with incrementing address, (8 words) : 15 cycles

4.5. OSD access

standard read access with incrementing address (16 words burst) : 23 cycles

4.6. Compressed data read for Start code detector

standard read access with incrementing address (16 words burst) : 23 cycles

4.7. Compressed data write

standard read access with incrementing address (16 words burst) : 23 cycles

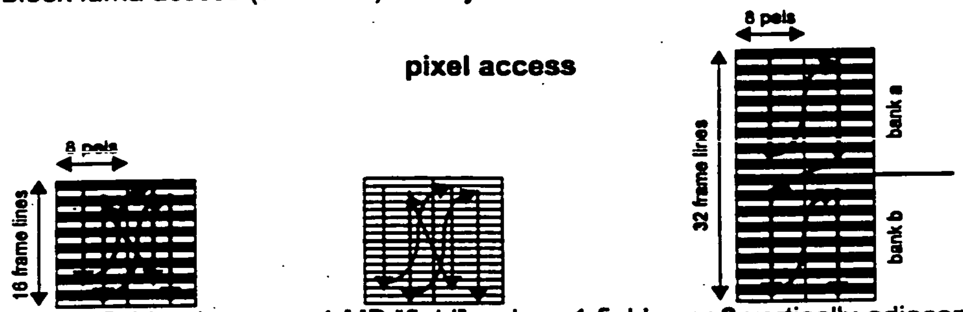
4.8. Compressed data read for VLD

standard read access with incrementing address (16 words burst) : 23 cycles

4.9. Reconstruction macroblock access

4.9.a. No compression mode

- 1 Macroblock luma access (32 words) : 39 cycles



1 MB field order (field pict)

1 MB "field" order (progressive frame)

1 field over 2 vertically adjacent MBs (MPEG 2 field structure pict)

Addressing :

Rn = row number of the accessed macroblock (vertical position of the accessed MB)
MBR = horizontal position of the accessed MB
bank = Rn mod 2

Addresses sequence (global address in specified bank from which row and column addresses are extracted) :

for frame structure picture (field or progressive) :

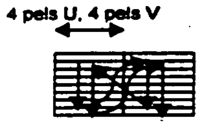
```
for k := 0 upto 1 { for i := 0 upto 3 {
add = $(Rn/2 \times W + MBR + FP) \times 32 + k \times 16 + i \times 4$ } }
```

for field structure picture, there is a bank access inside the burst.
Rn is always even (2 rows are reconstructed together), field is derived from the decoding instruction. So the calculation is :

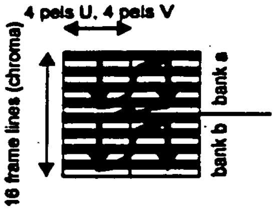
```
for j := 0 upto 1 { for i := 0 upto 3 { bank j selected
add = $((Rn/2) \times W + MBR_j + FP) \times 32 + field \times 16 + i \times 4$ } }
```

- 1 Macroblock chroma access (16 words) : 23 cycles

pixel access



1 MB frame
1 field after the other
(frame structure picture)



1 field over 2 vertically adjacent MBs
(MPEG 2 field structure pict)

Addresses sequence (global address in specified bank from which row and column addresses are extracted) :

for frame strucure picture :

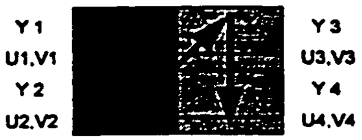
```
for k := 0 upto 1 { for i := 0 upto 1 {
add = $(FP + (FS+1)/2 + FA) \times 32 + (Rn/2 \times W + MBR) \times 16 + k \times 8 + i \times 4$ } }
```

for field structure picture, there is a bank access inside. field is derived from decoding instruction, so :

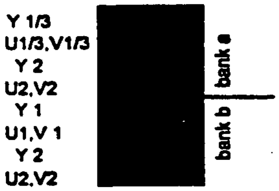
```
for j := 0 upto 1 { for i := 0 upto 1 { bank j selected
add = $(FP + (FS+1)/2 + FA) \times 32 + (Rn/2 \times W + MBR) \times 16 + field \times 8 + i \times 4$ } }
```

4.9.b. Mode 2M/3 (33% compression)

- 1 Macroblock access (32 words) : 39 cycles



1 MB
(frame structure picture)



same field over 2 vertical adjacent MB
(field structure picture)

Addresses sequence (global address in specified bank from which row and column addresses are extracted) :

bank = Rn mod 2

for frame structure picture :

```
for k := 0 upto 1 { for i := 0 upto 3 {
add = (Rn/2 x W + MBR + FP) x 32 + k x 16 + i x 4 } }
```

for field structure picture, there is a bank access inside. field is derived from decoding instruction, so :

```
for j := 0 upto 1 { for i := 0 upto 3 {
add = (Rn/2 x W + MBR + FP) x 32 + field x 16 + i x 4 } }
```

4.9.c. Mode M/2, H/2 (75% compression)

1 Macroblock chroma access (8 words) : 15 cycles.

Same accesses than no-compression mode concerning the blocks.

Addresses sequence :

bank = Rn mod 2

for frame structure picture :

```
for k := 0 upto 1 { add = FP x 32 + (Rn/2 x W + MBR) x 8 + k x 4 }
```

for field structure picture, there is a bank access inside, field is derived from decoding instruction :

```
for j := 0 upto 1 { bank = j ; add = FP x 32 + (Rn/2 x W + MBR) x 8 + field x 4 }
```

- 1 Macroblock chroma frame structure access (4 words) : 11 cycles

add = (FP + FA) x 32 + ((FS+1)/2x2 + Rn/2 x W + MBR) x 4 }

- 1 Macroblock chroma field structure access (4 words) : 14 cycles

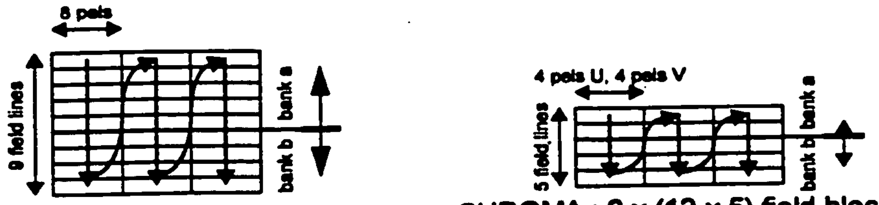
```
for j := 0 upto 1
{ bank = j ; add = (FP + FA) x 32 + ((FS+1)/2x2 + Rn/2 x W + MBR) x 4 + field x 2 }
```

4.10. Prediction macroblock access

4.10.a.No compression mode

Macroblocks of luma and chroma are store in such a way that the prediction is never over 2 pages for both luma and chroma. For this purpose, FA (frame pointer adjustment) is used to obtain the following equation true : $1 \leq (FA + (FS+1)/2) \bmod 8 \leq 6$

Prediction access for one field predictor



LUMA : 24 x 9 field blocks in 3 columns

CHROMA : 2 x (12 x 5) field blocks in 3 columns

The basic burst for luma is a 24x9 pixels field block. To generate a single luma prediction of a macroblock (forward or backward, field or frame), two burst are needed. (for frame prediction, the two burst will access both fields on the same location of the predictive picture). To generate a bidirectional prediction, field or frame, two bursts are needed from each anchor frame, so 4 burst access are necessary.

The basic burst for chroma is a 2x(12x5) pixels field block, one 12x5 block for each U and V. The prediction requirements are the same than for luma.

Address of the first word to be accessed :

Rn = vertical position of the macroblock to be predicted (MB in the pipeline)
MBR = horizontal position of this macroblock
Vx = half pel horizontal prediction vector (1/2 pixel resolution)
Vy = half pel vertical prediction vector in the frame (1/2 pixel resolution)
FP = frame buffer base address of the anchor frame from which the prediction is extracted
field = depends on the type of prediction, the vertical vector and the burst
/ means integer division with rounding towards "$-\infty$"
whereas DIV means integer division with rounding towards "0"

for luma :
$\text{bank} = (Rn + Vy/32) \bmod 2$
$\text{add} = (FP + (Rn + Vy/64) \times W + MBR + Vx/32) \times 32 + \text{field} \times 16$
$+ ((Vx/16) \bmod 2) \times 8 + (Vy/4) \bmod 8$

for choma :
$Vxc = Vx \text{ DIV } 2$, half pel horizontal prediction vector for chroma
$Vyc = Vy \text{ DIV } 2$, half pel vertical prediction vector for chroma in frame
$\text{bank} = (Rn + Vyc/16) \bmod 2$
$\text{add} = (FP + (FS+1)/2 + FA) \times 32 + ((Rn + Vyc/32) \times W + MBR + Vxc/16) \times 16 + \text{field} \times 8$
$+ ((Vxc/8) \bmod 2) \times 4 + (Vyc/4) \bmod 4$

Different characteristics of the bursts are described below.

Maximum length :

1 Macroblock luma field predictor access (27 words) without page boundary access inside : 36 cycles.
1 Macroblock luma field predictor access (27 words) with page boundary access inside : 42 cycles.
1 Macroblock chroma field predictor access (15 words) without page boundary access inside : 24 cycles.
1 Macroblock chroma field predictor access (15 words) with page boundary access inside : 30 cycles.

Minimum length :

1 Macroblock luma field predictor access (27 words) without page boundary access inside : 34 cycles.
1 Macroblock luma field predictor access (27 words) with page boundary access inside : 37 cycles.
1 Macroblock chroma field predictor access (15 words) without page boundary access inside : 23 cycles.
1 Macroblock chroma field predictor access (15 words) with page boundary access inside : 29

cycles.

The phase adjust is done on beginning of process if length is odd. If an odd number of memory 64-bits words are accessed, then one half of the first or last 128-bit DBusRead word for the prediction fifo is garbage (but it won't be used).

4.10.b.Mode 2M/3 (33% compression)

Prediction access for one field predictor

Y
U,V
Y
U,V
bank a
bank b

LUMA AND CHROMA : 6 adjacent blocks of the picture of same field
in 3 columns
8x8 blocks for luma, 4x4 blocks for chroma

The basic burst is 6 adjacent field blocks for luma, chroma U and chroma V. If the frame is a progressive frame with frame blocks compressed and store in the memory, then the 6 blocks correspond to 3 horizontally adjacent blocks on the same position over two rows of macroblocks (i.e. top blocks or bottom blocks of two rows), to keep the same access type than in field picture.

To generate a single luma prediction of a macroblock (forward or backward, field or frame, field or progressive picture), two burst are needed. (for frame prediction, the two bursts will access both fields on the same location in field picture or top and bottom blocks of the rows of macroblocks in progressive picture). To generate a bidirectional prediction, two bursts are needed from each anchor frame, so 4 burst access are necessary.

1 Macroblock field predictor access, luma and chroma on the same burst (36 words luma, 12 words chroma) :
- with/without page access inside : 55 cycles

Address of the first word to be accessed :

With same notations as above.

bank = (Rn + Vy/32) mod 2
add = (FP+ (Rn+Vy/64)xW + MBR + Vx/32)x32 + fieldx16 + ((Vx/16) mod 2)x8

4.10.c.Mode M/2, H/2 (75% compression)

As for no-compression mode, macroblocks of luma and chroma are store in such a way that the prediction is never over 2 pages for both luma and chroma. For this purpose, FA (frame pointer adjustment) is used to obtain the following equation true :
$1 \leq (4xFA + (FS+1)/2) \bmod 32 \leq 31$

The same type of data as 2M/3 mode are accessed, but luma and chroma are accessed separately. U and V chroma are accessed together.

1 Macroblock luma field predictor access (12 words) without page boundary access inside : 22 cycles.

1 Macroblock luma field predictor access (12 words) with page boundary access inside : 32 cycles.
1 Macroblock chroma field predictor access (6 words) without page boundary access inside : 17 cycles.
1 Macroblock chroma field predictor access (6 words) with page boundary access inside : 27 cycles.

Address of the first word to be accessed :

With same notations as above.

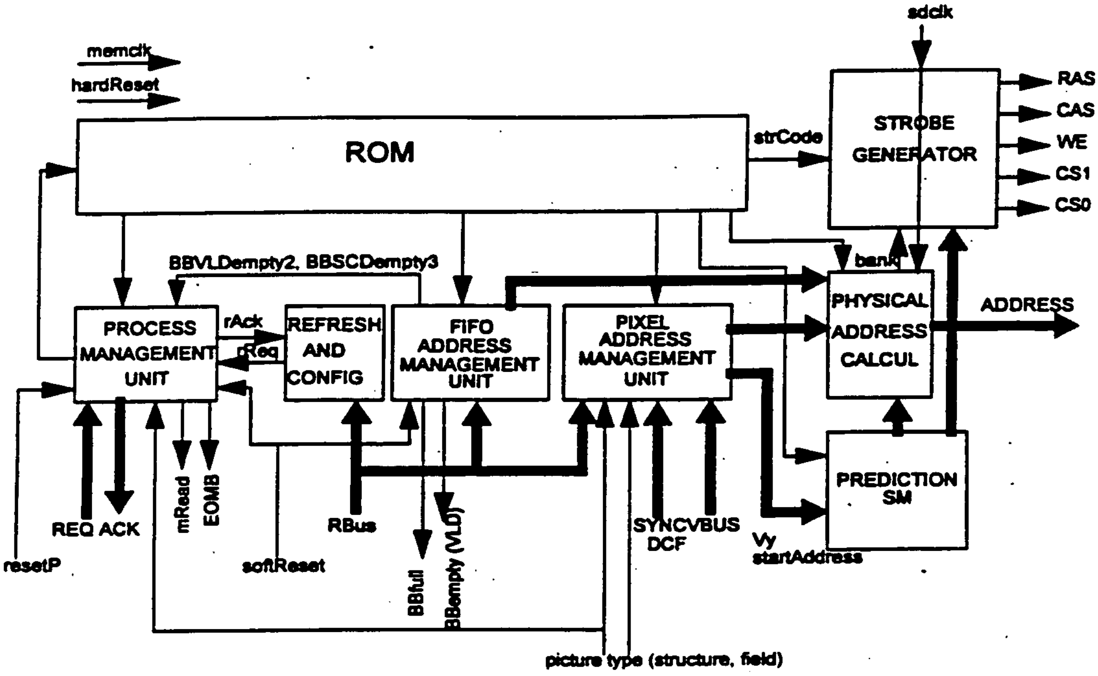
for luma :
bank = (Rn + Vy/32) mod 2
add = FPx32 + ((Rn + Vy/64)xW + MBR + Vx/32) x 8 + field x 4 + ((Vx/16) mod 2) x 2

for choma :
bank = (Rn + Vyc/16) mod 2
add = (FP+FA)x32 + (2x(FS+1)/2 + (Rn+Vyc/32) x W + MBR +
(Vxc/16) x 4 + field x 2+ (Vxc/8) mod 2

4.11. Block copy

128 bit read access followed by 128 bit write access : 20 cycles

5. LMC Architecture


memclk
hardReset
ROM
strCode
sdclk
STROBE GENERATOR
RAS
CAS
WE
CS1
CS0
bank
BBVLDempty2, BBSCDempty3
PROCESS MANAGEMENT UNIT
rAck
rReq
REFRESH AND CONFIG
FIFO ADDRESS MANAGEMENT UNIT
PIXEL ADDRESS MANAGEMENT UNIT
PHYSICAL ADDRESS CALCUL
ADDRESS
PREDICTION SM
REQ ACK
mRead
EOMB
resetP
softReset
RBus
BBfull
BBempty (VLD)
SYNCVBUS
DCF
Vy
startAddress
picture type (structure, field)

HD MPEG VIDEO DECODER

APPENDIX M

MOTION COMPENSATION

Thomson Consumer Electronics, Inc.
Indianapolis, Indiana, USA

Revision No. 2.2

THESE DRAWINGS AND SPECIFICATIONS ARE THE PROPERTY OF THOMSON CONSUMER ELECTRONICS, INC. AND SHALL NOT BE REPRODUCED OR COPIED OR USED AS THE BASIS FOR THE MANUFACTURE OR SALE OF APPARATUS OR DEVICES WITHOUT WRITTEN PERMISSION.

TCE PROPRIETARY AND CONFIDENTIAL

HD MPEG IC MOTION COMPENSATION DESIGN SPECIFICATION

Version : Version 3.1 - Friday, November 22, 1996, 2:09 pm
Author : John W. Gyurek - Jean-Michel Moutin

Revision History:

October 24, 1995: Creation
Monday, April 01, 1996 2:44 pm:Add information (JMM)
Monday, April 01, 1996 2:44 pm:Complete output stage description (JMM)
Monday, April 01, 1996 2:44 pm:Add info on middle stage (dec and reorder fifos) (JMM)
Monday, April 01, 1996 2:44 pm:complete middle stage (dec and reorder fifos) (JMM)

1. Introduction

This document describes the function and implementation of the motion compensation block of the HD MPEG IC. This block includes as datapath subsections the forward and backward interpolation filters, and the forward/backward summing block. The datapath blocks are controlled a master controller which has slave controllers at the input and output interface.

This block is responsible for doing the calculation of the predictor values that will be fed downstream to be combined with the decoded error or correction values that form the component video pixel data. The prediction values are zero for intra type macro-blocks. For cases of direct copies from the previous frame or field the down stream error value is zero and the motion vector is zero resulting in data from the same spacial area being fetched and fed downstream. For integer motion vectors a copy of data from a different spacial area is output. For non-integer motion vectors pixels adjacent to the desired location are averaged. These predictions can be backwards (from a future field or frame) forwards (from a previous field or frame) or both (bidirectional). If bidirectional prediction is used the forward and backward values are averaged.

2. Type of prediction

2.1. Intra, forward, backward, bidirectional prediction

The prediction pixels for a macroblock can be extracted from different reference images. For an Intra macroblock, there is no prediction pixels or the prediction can be seen as a macroblock of "0" pixels.

- Forward prediction: the prediction macroblock is generated from a previous reference frame only.
- Backward prediction: the prediction macroblock is generated from a future reference frame only (display in the future but already present in the external DRAM associated to the decoder).
- Bidirectional prediction: the prediction macroblock is generated as average of a forward prediction and a backward prediction: it is an average of a prediction extracted from a previous and from a

future frame.

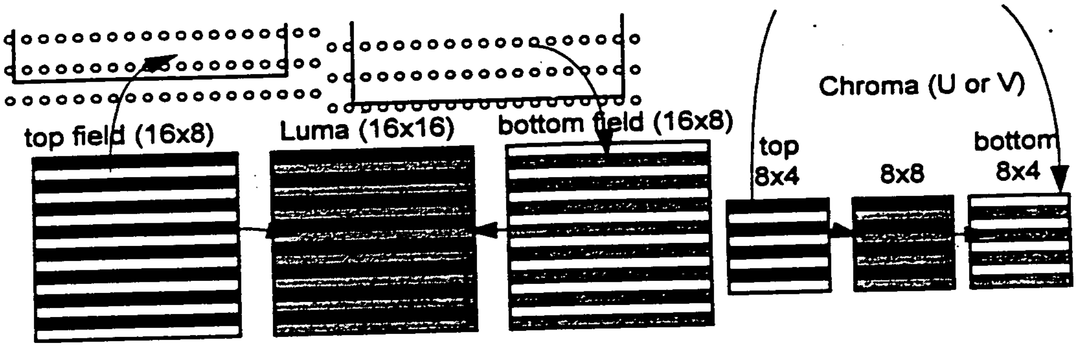
In the motion compensation unit, both directions of prediction (forward or backward) have there own path. Data entering a specific path (through the input fifos) is confined to this path.

When prediction is bidirectional, then both forward and backward predictors are of same type. Following are the description of the different type of prediction.

The predictor pixels can be direct pixels of a reference frame or average of neighbour pixels (if half pixel vector).

2.2. *Frame structure picture*

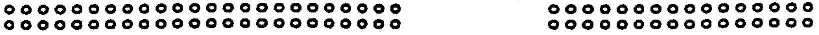
- frame prediction


top field (16x8)
Luma (16x16)
bottom field (16x8)
Chroma (U or V)
top
8x4
8x8
bottom
8x4

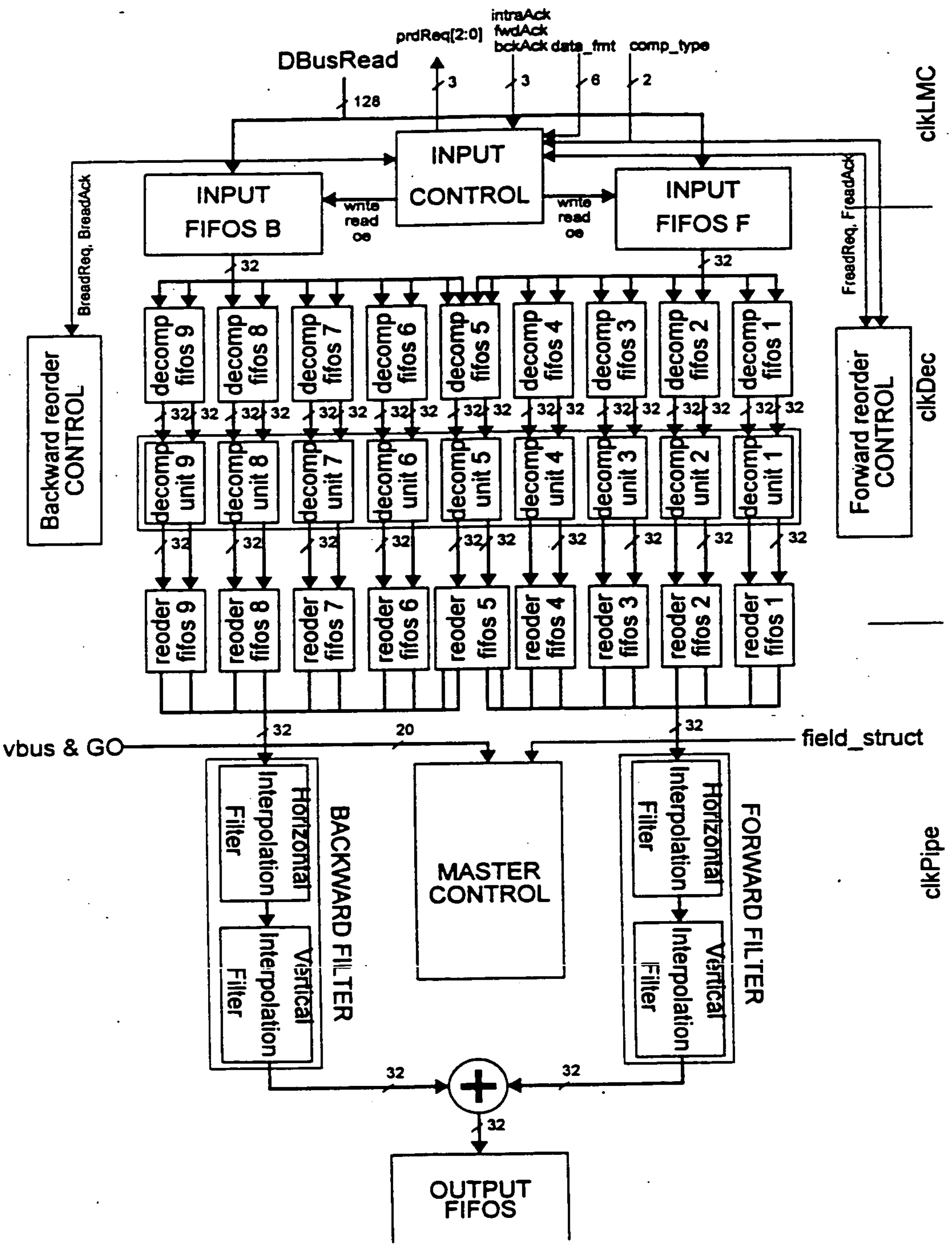
2.3. *Field structure picture*

- 16x16 prediction

Pixels of reference field (pixel belong to one field, not a frame)



3. Block diagram


DBusRead
128
prdReq[2:0]
3
intraAck
fwdAck
bckAck
3
data_fmt
6
comp_type
2
clkLMC
INPUT CONTROL
INPUT FIFOS B
INPUT FIFOS F
write
read
oe
write
read
oe
BreadReq, BreadAck
FreadReq, FreadAck
32
32
decomp fifos 9
decomp fifos 8
decomp fifos 7
decomp fifos 6
decomp fifos 5
decomp fifos 4
decomp fifos 3
decomp fifos 2
decomp fifos 1
Backward reorder CONTROL
Forward reorder CONTROL
clkDec
decomp unit 9
decomp unit 8
decomp unit 7
decomp unit 6
decomp unit 5
decomp unit 4
decomp unit 3
decomp unit 2
decomp unit 1
reoder fifos 9
reoder fifos 8
reoder fifos 7
reoder fifos 6
reoder fifos 5
reoder fifos 4
reoder fifos 3
reoder fifos 2
reoder fifos 1
32
vbus & GO
20
32
field_struct
BACKWARD FILTER
Horizontal Interpolation Filter
Vertical Interpolation Filter
MASTER CONTROL
FORWARD FILTER
Horizontal Interpolation Filter
Vertical Interpolation Filter
clkPipe
32
32
32
OUTPUT FIFOS

4. Description of interfaces

4.1. Clock and Clock control

- clkLMC – LMC clock (typically 50 MHz)
- clkDec – decompression clock (typically 81 MHz)
- clkPipe – pipe clock (typically 54 MHz)

4.2. LMC and memory interface

4.2.a. Data

- DbusRead[127:0] : 128 bits, coming from external SDRAM through DBus interface. Input data of the motion compensation unit

4.2.b. LMC handshake signals

- PRDreq[2:0] : request bus to the local memory controller (LMC). Request is non zero when input fifos of the motion compensation unit (MCU) can receive the next memory access and when this access defined (i.e. for the first access of a macroblock, when the data for this macroblock have been received by the MCU).

Table 1: LMC REQUEST CODING

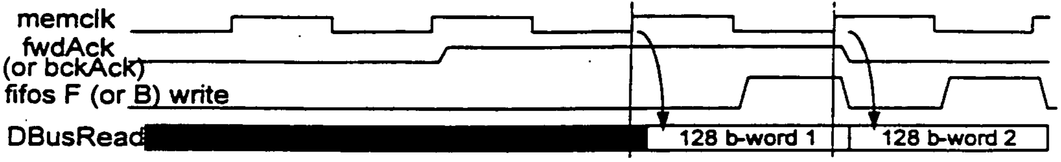
PRDreq[2:0]	REQUEST DESCRIPTION	
	frame structure picture	field structure picture
000 (0)	no request	
001 (1)	intra macroblock	
010 (2)	Forward progressive	Forward 16x16
011 (3)	Forward field	Forward 16x8
100 (4)	Backward progressive	Backward 16x16
101 (5)	Backward field	Backward 16x8
110 (6)	Bidirectional progressive	Bidirectional 16x16
111 (7)	Bidirectional field	Bidirectional 16x8

Acknowledges signals are answer from the LMC to the request.

- intraAck : Asserted during one lmc clock cycle when intra macroblock is going on the pipeline
- fwdAck : Asserted when DBusRead contains data to be received into forward input fifos
- bckAck : Asserted when DBusRead contains data to be received into backward input fifos
- burstEnv : this signal is high during a whole burst of data coming from the LMC (i.e. during a burst, fwdAck or bckAck may fall and rise again when invalid data are on DBusRead, burstEnv stays high as long as data belong to the same burst. This signal represent the envelopp of the fwdAck or bckAck signal for the burst)

- data_fmt is a 6-bits bus, active during no compression access with the acknowledge signal, and indicates what words are dummy words written to the fifo.

Timing diagram of LMC interface



4.2.c. Additional static bit information

- comp_type : 2 bit for memory compression type used

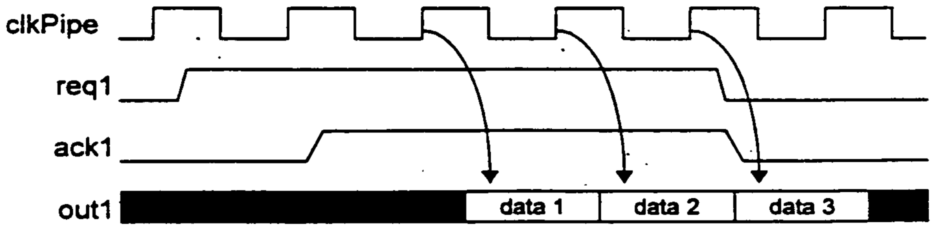
00 means no compression
01 means 2M/3 compression
11 means H/2-M/2 compression

- field_struct : when high indicates that the structure of the picture is field (otherwise it is frame).

4.3. Output interface

- req1 : request from the MCU to the adder 1, means the MCU is ready to send data through out1
- ack1 : acknowledge from the adder 1 to the MCU, means data from out1 will be read on the next edge of the clock
- out1[7:0] : 8 bits, prediction pixel for adder 1 in raster scan order
- req2 : request from the MCU to the adder 1, means the MCU is ready to send data through out1
- ack2 : acknowledge from the adder 1 to the MCU, means data from out1 will be read on the next edge of the clock
- out2[7:0] : 8 bits, prediction pixel for adder 2 in raster scan order

Output interface timing diagram (interface to adder 1 or 2 are the same)



4.4. VLD bus signals

The VLD (variable length decoder) communicates with the MCU via a bus system (vBusA, vbusD) plus a control signal (GO) and an enable signal (vbusAck).

- GO – Asserted by the VLD when all macroblock data is valid and has been sent. Allows the MCU to start processing the data.

This signal should be used to strobe the motion vectors and the macroblock type data into a shadow registers.

- vbusA[3:0] – Address bus from VLD
- vbusD[13:0] – Data bus from VLD (not all bits are used in the MCU)
- vbusAck – Acknowledge signal: other signals of the VLD interface only have meaning when this signal is high. If low, this interface is disable (also GO).

The MCU responds to the following addresses:

Table 2: VLD BUS ADDRESSES

ADDRESS	SIGNAL on vbusD	DESCRIPTION
00H	HFO	Forward horizontal vector (top field if field prediction or upper blocks if 16x8)
01H	VFO	Forward vertical vector (top field if field prediction of upper blocks if 16x8)
02H	HFE	Forward horizontal vector bottom field or lower blocks
03H	VFE	Forward vertical vector bottom field or lower blocks
04H	HBO	Backward horizontal vector (top field if field prediction or upper blocks if 16x8)
05H	VBO	Backward vertical vector (top field if field prediction or upper blocks if 16x8)
06H	HBE	Backward horizontal vector (bottom field if field prediction or lower blocks if 16 x 8)
07H	VBE	Backward vertical vector (bottom field or lower blocks if 16x8)
0CH	MType	Macroblock type (information on prediction type)

Vbus vector data (for vbusA 0 to 7) are 12-bits words on vbusD[11:0] and correspond to a 2's complement representation of motion vector component in half pels.

Not all the 12-bits are necessary to be store in the MCU for its job.

MType is a 4 bits word on vbusD[3:0]. Its meaning is the following:

The Mtype[3:0] values are defined as:

Table 3: MTYPE[3:0] MACROBLOCK TYPE VALUES

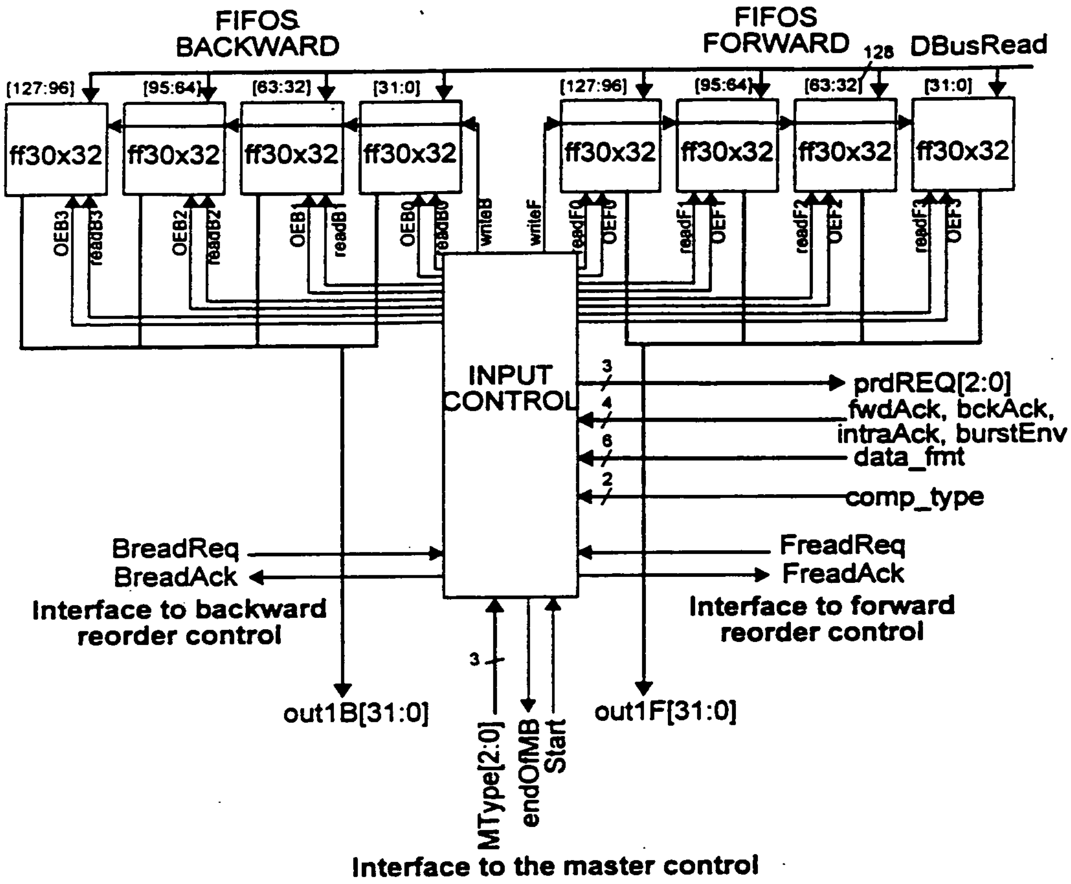
Value	Macroblock type
0000 (0)	Not used
0001 (1)	Intra type macro block (no predictors used)

Table 3: MTYPE[3:0] MACROBLOCK TYPE VALUES

Value	Macroblock type
0010 (2)	forward progressive (frame structure) of forward 16X16 (field structure) predicted macroblock
0011 (3)	forward field (frame structure) or forward 16x8 (field structure) predicted macroblock
0100 (4)	backward progressive (frame structure) or backward 16x16 (field structure) predicted macroblock
0101 (5)	backward field (frame structure) or backward 16x8 (field structure)
0110 (6)	bidir progressive (frame structure) or bidir 16x16 (field structure)
0111 (7)	bidir field (frame structure) or bidir 16x8 (field structure)
1000 (8)	copy predicted: forward progressive with HFO = VFO forced to 0
1001 (9)	copy bidirectional: same type as preceding macroblock, but frame type prediction.

If Mtype = 9 is received by the MCU, the internal macroblock type of the MCU is derived from the previous one by forcing the LSB to "0", except if the previous was "1" (intra) then the new one is forced to "2".

5. *Input Fifos description*



5.1. *Input control general function*

When Input control receives a 'start' signal from the master control, it generates request to the LMC (prdReq) taking in account MType and the available space in the input fifos. The input control will then receive an acknowledge from the LMC and write the data on the DBusRead to the appropriate set of fifos (F or B). The amount of data written to the fifos is equal to a memory burst. When all the writing has been done for the current macroblock, the "endOf MB" line to the master control is asserted.

5.2. *Memory data input bursts*

The clock of the input interface is clkLMC.

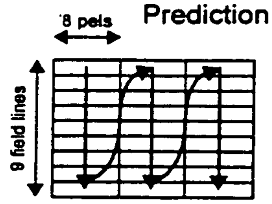
Several burst memory accesses are done to generate one predictor. The order of the bursts are:

- no compression mode:

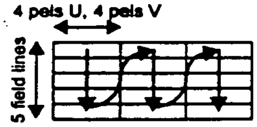
luma burst 1 (14 or 15 128-bits words)

luma burst 2 (14 or 15 128-bits words)
chroma burst 1 (8 or 9 128-bits words)
chroma burst 2 (8 or 9 128-bits words)

Prediction access for one field predictor



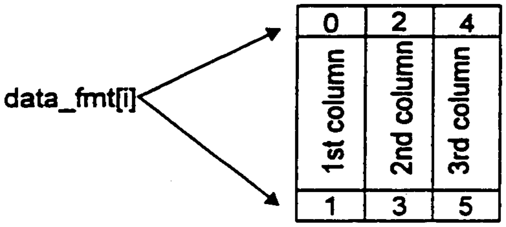
LUMA burst: 24 x 9 field blocks
in 3 columns



CHROMA burst: 2 x (12 x 5) field blocks
in 3 columns

An 8 pixel word corresponds to a 64-bit word. Two such words are concatenated to generate a 128-bit word.

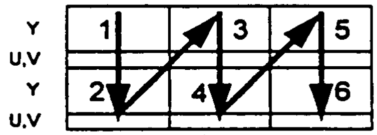
The 6-bits data_fmt bus indicates the words that are written to the fifos but do not correspond to a real data read from the memory (i.e. words added for the demux purpose). Those words can only be at the top or the bottom of a 9 line column. data_fmt[i] is set when the corresponding word on the access is a dummy word:



- 2M/3 mode:

burst 1 (24 128-bits words)
burst 2 (24 128-bits words)

Prediction access for one field predictor



LUMA AND CHROMA burst: 6 adjacent blocks of the picture of same field
in 3 columns
8x8 blocks for luma, 4x4 blocks for chroma

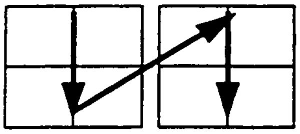
Numbering of the blocks as shown inside will be used further in the spec. The second burst is numbered the same way from 7 to 12. The chroma blocks following a luma block have the same number.

Here as well 2 following 64-bit words extracted from the memory are a 128-bit DBusread word. No dummy words are written in this mode.

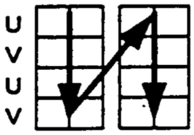
- H/2M/2 mode:

luma burst 1 (8 128-bits words)
luma burst 2 (8 128-bits words)
chroma burst 1 (4 128-bits words)
chroma burst 2 (4 128-bits words)

Prediction access for one field predictor



LUMA: 8 adjacent blocks (4 M/2 blocks) of same field in 2 columns



CHROMA: 8 adjacent blocks of U and V (4 M/2 block of U and V) of same field in 2 columns

In all modes, when a bidirectional macroblock is on process, forward and backward burst alternate, starting with a froward access.

5.3. *Input controller output data read description*

Both output interfaces (F or B) are equivalent. The clock of this interface is clkDec. The following description is for one interface only (F or B).

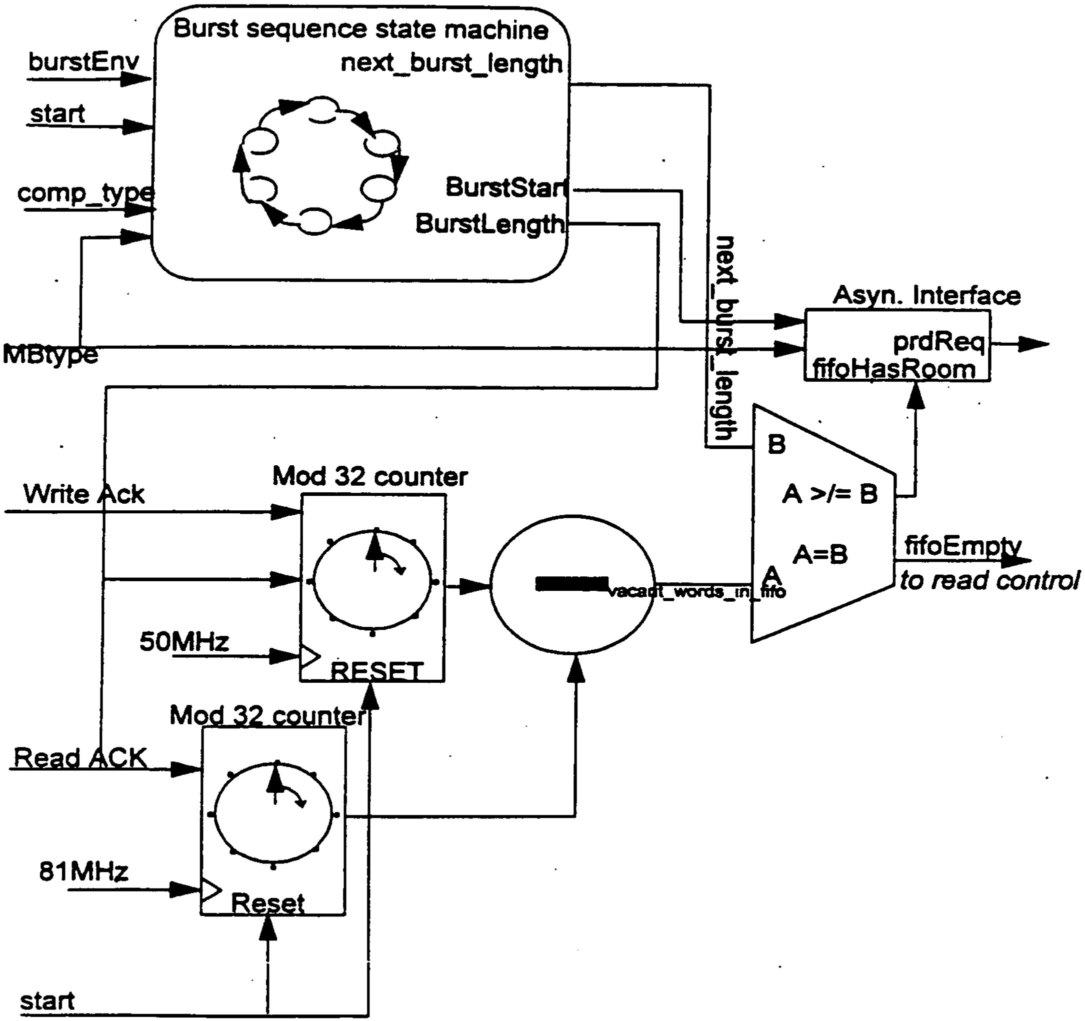
When the middle stage of the MCU is ready to receive a data, readReq is asserted. The input controller reads the input fifo in a raster order within a column and asserts readAck. The fifo read order is: 0,1,2,3 (MSB to LSB in the fifo block diagram). The output rate is 81 Mhz: the read is done every clock cycle. When reading data corresponding to a dummy word, readAck is deasserted causing the middle stage to ignore the non-valid data.

Note: for one direction (F or B), data_fmt must be stored for 3 bursts that can be pipelined in the input fifo.

The output read order corresponds to the input read order, except that it is in words of 32 bits. (MORE DETAIL NEEDED!)

5.4. *Input controller architecture*

The input controller is compossed of two state machines and some special asynchronous interfaces. This is necessary because the fifo input is running at the LMC clock rate of 50MHz and the output to the decompression or re-ordering fifos runs at 81MHz. A block diagram is shown below:



This block diagram shows the opperation of the input fifo controller. The Burst sequence state machine keeps track burst sequence order for each of the types of prediction and compression. In doing this the length of the next write burst is know. The difference of the read and write counters indicates how much room is in the fifo. IThe comparator will indicate if there is room for the next burst sequence. If there is room then prdReq will take on the non-zero value of the MBtype else, prdReq will be held low.

The level of the fifo is calculated using two counters a subtracter and a comparator. Because of the uncertanty of the burst length in no compression mode the fifo is controled as if it were 30 bits in length when it is realy 34 words long.

6. Middle stage description

This stage is used to reoder and/or decompress data coming from the memory to make them suitable for filtering and interpolation. If decompression has to be used, then data read from the input fifos goes through decompression fifos and decompression units, and is then written to the reordering fifos. When no decompression is used, data goes directly from the input fifos to the reordering fifos.

The middle stage receives 24x9 windows (no compression mode), 24x16 windows (2M/3 compression mode) or 32x16 windows (H/2-M/2 compression) from the input fifos but only writes 20x9 windows in the reordering fifos: part of the horizontal and vertical vectors are used for this purpose.

6.1. Decompression fifos

The decompression fifos get data from the input fifos and fill the decompression units. The write order is driven by the middle stage controller. The read is controlled locally with the associated decompression unit.

Decompression fifos 1 to 3 and 6 to 8 are composed of 2 fifos, each 24 words of 32 bits. Decompression fifos 4 and 9 are composed of 2 fifos, each 16 words of 32 bits. Every set of fifos has its own empty/full controller. The decompression fifo 5 is composed of 4 fifos of 8 words of 32 bits, with a controller too, 2 fifos are associated with the forward datapath, 2 with the backward one.

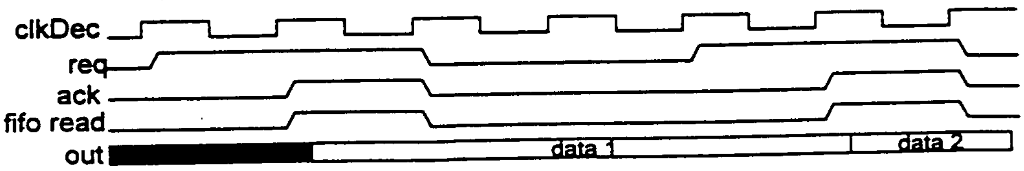
The decompression fifos and controller run at clkDec (81 MHz).

6.1.a. Decompression fifos controllers

- Decompression fifos 1 to 4 or 6 to 9 controller

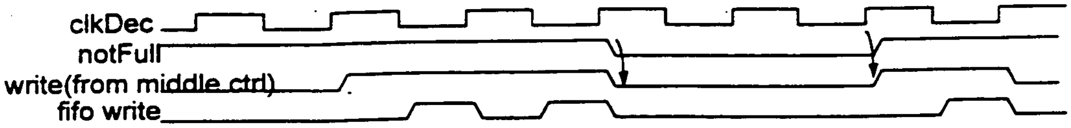
Those decompression fifo controller receive 2 read request from the associated decompression unit and a 2 write signals from the middle stage controller. They generate read and write pulses to the two fifos. They generate notFull signal for everfy fifo that they send to the middle stage controller. They internally generate empty signal that they use to generate ack to the decompression unit, as well as the request. The output of each fifo is connected to a different input of the decompression unit.

Decompression fifos output timing (the rate is driven by the decompression need) :



There can be only one word read at a time (never 2 following words). The oe signal of the fifos is always high.

Decompression fifos intput timing (fifos here are written at 81 MHz)

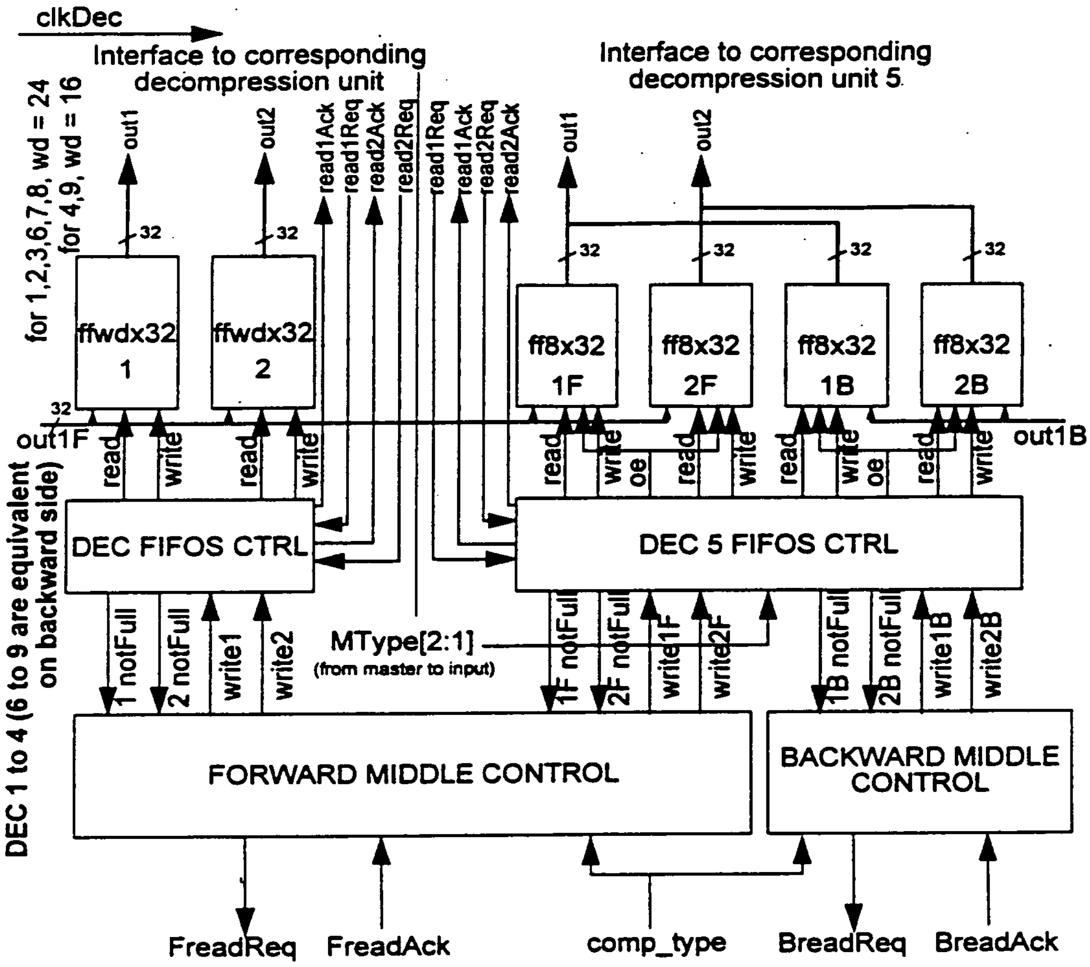


The write signal from middle control is derived from a sequence known internally to the middle control and the notempty signal.

- Decompression fifos 5

This set of decompression fifo behave the same as the other except that it contains 4 fifos instead of 2. So 2 fifos are able to write to the same input of the decompression unit 5. So this request the management of the oe signal of the fifos for multiplexing reasons. This management will be explained later.

6.1.b. Decompression fifos general block diagram



The two fifos of a set contain different compressed blocks, that will be decompressed in an interleaved way in the associated decompression unit. The request of data for these two blocks are independent (due to the compression scheme), that is why 2 output buses and request/acknowledge signals are necessary between fifo set and decompression unit.

The decompression 5 fifo is different from the others because it is shared between both F and B directions. 2 fifos are dedicated to both directions. The decompression unit will work for just one at a time, but will toggle then (though the oe pins). The DEC 5 control will control that.

The "notFull" flag is sent by every fifo controller to the middle controller for each of its fifos.

6.2. Middle control

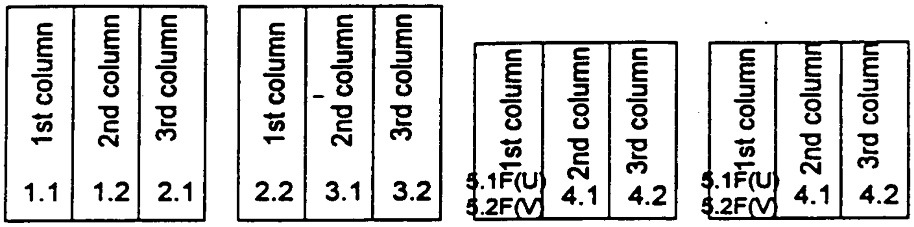
There are two symmetrical middle control: forward and backward. Their purpose is to select the direction of data coming out of input fifos. Description is made of the forward middle controller. The backward is identical.

Depending on compression mode the task is not the same.

6.2.a. No compression

Decompression unit are in bypass mode.

The burst are sent to the fifos according to the following picture. The number inside the column means "decompression_set_number.fifo_number".



luma 1st burst | luma 2nd burst | chroma 1st burst (alternate U and V) | chroma 2nd burst (alternate U and V)

The order of write for a macroblock is then :

- 18 writes to fifo 1.1
- 18 writes to fifo 1.2
- 18 writes to fifo 2.1
- 18 writes to fifo 2.2
- 18 writes to fifo 3.1
- 18 writes to fifo 3.2
- 10 alternates writes to fifos 5.1F and 5.2F, starting with 5.1F
- 10 writes to fifo 4.1
- 10 writes to fifo 4.2
- 10 alternates writes to fifos 5.1F and 5.2F, starting with 5.1F
- 10 writes to fifo 4.1
- 10 writes to fifo 4.2

Total : 168 writes

Decompression unit are in bypass mode.

For the backward side, 5.1B is equivalent to 5.1F, 5.2B to 5.2F, and fifo set 6 is equivalent to set 1, 7 to 2, 8 to 3, 9 to 4.

6.2.b. 2M/3 compression

- The burst are sent to the fifos according to the following picture. The number inside the column means "decompression_set_number.fifo_number (block_number)".

Y	1.1(Y1)	1.2(Y3)	2.1(Y5)
U	5.1F(U1)	4.1(U3)	4.2(U5)
V	5.2F(V1)	4.1(V3)	4.2(V5)
Y	1.1(Y2)	1.2(Y4)	2.1(Y6)
U	5.1F(U2)	4.1(U4)	4.2(U6)
V	5.2F(V2)	4.1(V4)	4.2(V6)

First burst

Y	2.2(Y7)	3.1(Y9)	3.2(Y11)
U	5.1F(U7)	4.1(U9)	4.2(U11)
V	5.2F(V7)	4.1(V9)	4.2(V11)
Y	2.2(Y8)	3.1(Y10)	3.2(Y12)
U	5.1F(U8)	4.1(U10)	4.2(U12)
V	5.2F(V8)	4.1(V10)	4.2(V12)

Second burst

The order of write for a macroblock is then :

- 12 writes to fifo 1.1
- 2 writes to fifo 5.1F
- 2 writes to fifo 5.2F
- 12 writes to fifo 1.1
- 2 writes to fifo 5.1F
- 2 writes to fifo 5.2F
- 12 writes to fifo 1.2
- 4 writes to fifo 4.1
- 12 writes to fifo 1.2
- 4 writes to fifo 4.1
- 12 writes to fifo 2.1
- 4 writes to fifo 4.2
- 12 writes to fifo 2.1
- 4 writes to fifo 4.2
- 12 writes to fifo 2.2
- 2 writes to fifo 5.1F
- 2 writes to fifo 5.2F
- 12 writes to fifo 2.2
- 2 writes to fifo 5.1F
- 2 writes to fifo 5.2F
- 12 writes to fifo 3.1
- 4 writes to fifo 4.1
- 12 writes to fifo 3.1
- 4 writes to fifo 4.1
- 12 writes to fifo 3.2
- 4 writes to fifo 4.2
- 12 writes to fifo 3.2
- 4 writes to fifo 4.2

Total : 192 writes

Decompression unit 1 decompress Y1 and Y3 interleaved, then Y2 and Y4.

Decompression unit 2 decompress Y5 and Y7 interleaved, then Y6 and Y8.
Decompression unit 3 decompress Y9 and Y11 interleaved, then Y10 and Y12.
Decompression unit 4 decompress U3 and U5 interleaved, then V3 and V5, U4 and U6, V4 and V6, U9 and U11, V9 and V11, U10 and U12, V10 and V12.
Decompression unit5 decompress U1 and V1 interleaved, then U2 and V2, then the same of the backward path, then U7 and V7, U8 and V8, then the same on the backward side. If prediction is only forward or backward, only the blocks of the prediction direction are decompressed.

For the backward side, 5.1B is equivalent to 5.1F, 5.2B to 5.2F, and fifo set 6 is equivalent to set 1, 7 to 2, 8 to 3, 9 to 4.

6.2.c. *H/2-M/2 compression*

The burst are sent to the fifos according to the following picture, with the same convention as above.

1.1(Y1)	1.2(Y3)
1.1(Y2)	1.2(Y4)

luma 1st burst

3.1(Y5)	3.2(Y7)
3.1(Y6)	3.2(Y8)

luma 2nd burst

U	4.1	4.2
V	4.1	4.2
U	4.1	4.2
V	4.1	4.2

chroma 1st burst

U	4.1	4.2
V	4.1	4.2
U	4.1	4.2
V	4.1	4.2

chroma 2nd burst

The order of write for a macroblock is then :

- 16 writes to fifo 1.1
- 16 writes to fifo 1.2
- 16 writes to fifo 3.1
- 16 writes to fifo 3.2
- 8 writes to fifo 4.1
- 8 writes to fifo 4.2
- 8 writes to fifo 4.1
- 8 writes to fifo 4.2

Total : 96 writes

Decompression unit 1 decompress Y1 and Y3 interleaved, then Y2 and Y4.
Decompression unit 3 decompress Y5 and Y7 interleaved, then Y6 and Y8.
Decompression unit 4 decompress U1 and U3 interleaved, then V1 and V3, U2 and U4, V2 and V4, U5 and U7, V5 and V7, U6 and U8, V6 and V8.
Decompression units 2 and 5 are not used.

For the backward side, 5.1B is equivalent to 5.1F, 5.2B to 5.2F, and fifo set 6 is equivalent to set 1, 7 to 2, 8 to 3, 9 to 4.

6.2.d. *FreadReq (BreadReq) and write order control*

The middlestage controller counts (with an 8 bit counter) the writes to the decompression fifos (through "readAck" signal). The "write" signals issues to the fifo controllers are function of the count and the compression type. The nextcount is used to know the next fifo to be written in and this signal is gated with the "notFull" of the appropriate fifo to generate the "readReq" to the input controller. The count restart to 0 when the number of writes for the actual compression type is reached.

6.2.e. Additional information for fifo controllers

If for a given macroblock prediction is only forward or backward, the fifo corresponding to the non used direction will not be filled. The associated decompression unit will then not run.

Only the decompression unit 5 has to care. This unit strobes the MType[2:1] value (sent to the input controller by the master controller) when starting the writes of a new macroblock in the fifos.

- If Mtype[2:1] = 1, macroblock is forward
- If Mtype[2:1] = 2, macroblock is backward
- If Mtype[2:1] = 3, macroblock is bidir

Internally, the MType is latched again when the macroblock start to be read by the decompression unit. This value is used to manage the read and oe of the fifos. This is used in no compression and 2M/3 modes (in H/2-M/2, unit 5 is not used). The read sequences are as follow :

- No compression mode :
  - forward macroblock : out1 : 10 reads of 5.1F ; out2 : 10 reads of 5.2F
  - backward macroblock : out1 : 10 reads of 5.1B ; out2 : 10 reads of 5.2B
  - bidirectional macroblock :
    out1 : 5 reads of 5.1F, 5 reads of 5.1B, 5 reads of 5.1F, 5 reads of 5.1B ;
    out2 : 5 reads of 5.2F, 5 reads of 5.2B, 5 reads of 5.2F, 5 reads of 5.2B
- 2M/3 compression mode :
  - forward macroblock : out1 : 8 reads of 5.1F ; out2 : 8 reads of 5.2F
  - backward macroblock : out1 : 8 reads of 5.1B ; out2 : 8 reads of 5.2B
  - bidirectional macroblock :
    out1 : 4 reads of 5.1F, 4 reads of 5.1B, 4 reads of 5.1F, 4 reads of 5.1B
    out2 : 4 reads of 5.2F, 4 reads of 5.2B, 4 reads of 5.2F, 4 reads of 5.2B
    In this case, unit 5 decompress (U1,V1,U2,V2) forward, then (U1,V1,U2,V2) backward, then (U7,V7,U8,V8) forward, and (U7,V7,U8,V8) backward.

6.3. Reordering fifos

6.3.a. General presentation and block diagram

The reordering fifos receive data from the decompression units and give data to the datapath. The write is done in each unit depending on decompression work. The read order is driven by the master controller. Input runs with clkDec (81 MHz), out with clkPipe (54 MHz). The fifo purpose is also asynchronous interface.

Reordering fifos 1, 2, 3, 6, 7and 8 are composed of 2 fifos, each 18 words of 32 bits.
Reordering fifos 4 and 9 are composed of 8 fifos, each 8 words of 32 bits.
Reordering fifo 5 is also composed of 8 fifos, each 6 words of 32 bits, 4 are associated with the forward datapath, 4 with the backward one. Every set of fifos has its own empty/full controller.

```
'Function macro: block_clks(    comp_mode, bus_width    )
'
'This function computes the number of cycles required to access a storage
'unit from memory.  The definition of a storage unit depends on the
'bus_width parameter.  The bus_widths supported for these calculations are
'64 and 128 only.  For 64 bit bus, the storage unit is one field-block.  For
'128 bit bus, it is two adjacent field blocks.
'comp_mode takes values 0, 1, 2 corresponding to no compression, two-thirds
'and m/2;h/2
'bus_width takes values 64 and 128
Function block_clks(c_mode, b_width)
    Select Case b_width
    Case 64
        luma_pix = 64
        chroma_pix = 16
    Case 128
        If c_mode = 0 Then
            luma_pix = 64
            chroma_pix = 16
        Else
            luma_pix = 128
            chroma_pix = 32
        End If
    End Select

    Select Case c_mode
    Case 0
        'no compression
        block_clks = (luma_pix * 8) / b_width + (chroma_pix * 2 * 8) / b_width
    Case 1
        'two-thirds compression
        block_clks = (luma_pix * 8 * 3) / (4 * b_width) + (chroma_pix * 2 * 8) / (2 * b_width)
    Case 2
        'M/2-H/2 compression
        block_clks = (luma_pix * 8) / (b_width * 4) + (chroma_pix * 2 * 8) / (b_width * 4)
    End Select

End Function
```

```
'Function macro: stu_per_mb(    bus_width    )
'
'This function computes the number of storage units in each macroblock
'The bus_widths supported for these calculations are 64 and 128 only.
'bus_width takes values 64 and 128
Function stu_per_mb(b_width)
    Select Case b_width
    Case 64
        stu_per_mb = 4
    Case 128
        stu_per_mb = 2
    End Select

End Function
```

APPENDIX V

HD DECODER MEMORY BANDWIDTH - SETS 1, 2, AND 3

HD Decoder IC memory Bandwidth estimation using NEC uPD 4516161 SDRAM

This sheet: Listing of unrestricted variables

Definitions:

Unrestricted variables: These are variable within specified range. Effects of variations are incorporated in spreadsheet

Restricted variables: Only specified values are allowed. Changing these have global implications, so each case is dealt with in its own spreadsheet

Parameter Block:		**Constants:**	
Unrestricted variables:		frame_pic_type:	0
Mem. Clock rate (MHz):	100	field_pic_type:	1
Mem. CAS Latency (clks):	3	frame_mv:	0
Input Frame Vert. Size:	1088	field_mv:	1
Input Frame Horz. Size:	1920	dual_prime:	2
1 page hit in (accesses):	4	field_16by8:	3
Input Bitrate (bits/s):	83886080	no_comp:	0
OSD Horz. Size:	1920	two_thirds:	1
OSD Vert. Size:	363	mby2_hby2:	2
OSD Mode (2 or 4 bit/pixel):	1		
OSD Resolution:	0		
Percentage of 4 mv MBs:	80	osd_mode_2:	0
Active Video Horz. Size:	1920	osd_mode_4:	1
Active Video Vert. Size:	1088	osd_res_f:	0
Total Video Horz. Size:	2400	osd_res_h:	1
Total Video Vert. Size:	1125	osd_res_t:	2
Display Frame Rate:	30		
On-chip display line stores:	0	yes:	1
SDRAM display line stores:	16	no:	0
Memory page size:	256		
External bus width (bits):	64		
Percentage of worst case MBs:	90		

```
'Function macro: Write_clks (   data_size, latency, page_hit, new_page,
'                               hide_head, hide_tail, precharge, transition  )
' Assuming use of precharge to terminate bursts
Function write_clks(data_size, latency, page_hit, new_page, hide_head, hide_tail, precharge, transition)
    page = 256
    over_head = 0
    over_tail = 0
    body = 0

    If hide_head = 0 Then
        'hide_head will be FALSE (=0) for read -> write transitions from
        'the same bank when burst mode is page.  So page mode read->write
        'transition on same bank will look like non-overlapping accesses.
        'This was made necessary because read and write routines don't
        'include bank switching as part of overhead calculations.  They
        'rely on the hide_head and hide_tail variables to provide this
        'mechanism
        If precharge = 1 Then
            over_head = over_head + 3   'precharge to activation overhead
        End If
        If new_page = 1 Then
            'A new page is deemed to occur everytime the LMU reads from
            'a new area of mem. or if the LMU was forced to terminate
            'the last burst of data using a precharge (in the absence of
            'a pending access request)
            over_head = over_head + 3   'Activation overhead
        End If
    ElseIf transition = 1 Then
        'Following applies to all burst modes and latencies except page
        'mode, where it applies only in cases where the two accesses are
        'from different banks
        over_head = over_head + 1   'read to write Hi-Z cycle overhead
    End If
    If hide_tail = 0 Then
        If latency = 1 Or latency = 2 Then
            over_tail = over_tail + 1 + 3   'last data to precharge to activation
        Else
            over_tail = over_tail + 2 + 3
        End If
    End If
    If page_hit = 1 Then
        'We are ignoring the potential for the LMU to hide this
        'overhead by servicing a request from the other bank
        If latency = 1 Or latency = 2 Then
            body = body + 1 + 3 + 3
        Else
```

```
            body = body + 2 + 3 + 3
        End If
    End If

    write_clks = over_head + over_tail + body + data_size

End Function
```

```
'Function macro: Read_clks (    data_size, latency, page_hit, new_page,
'                               hide_head, hide_tail, precharge, transition  )
' Assuming use of precharge to terminate bursts
Function read_clks(data_size, latency, page_hit, new_page, hide_head, hide_tail, precharge, transition)
    page = 256
    over_head = 0
    over_tail = 0
    body = 0

    If hide_head = 0 Then
        If precharge = 1 Then
            over_head = over_head + 3   'precharge to activation overhead
        End If
        If new_page = 1 Then
            'A new page is deemed to occur everytime the LMU reads from
            'a new area of mem. or if the LMU was forced to terminate
            'the last burst of data using a precharge (in the absence of
            'a pending access request)
            over_head = over_head + 3   'Activation overhead
        End If
        over_head = over_head + latency 'Also applies to write->read xsition
    ElseIf transition = 1 Then
        ovvr_head = over_head + 1       'write to read Hi-Z cycle overhead
    End If
    If hide_tail = 0 Then
        over_tail = over_tail + 3
    End If
    If page_hit = 1 Then
        'We are ignoring the potential for the LMU to hide this
        'overhead by servicing a request from the other bank
        body = body + 3 + 3 + latency
    End If

    read_clks = over_head + over_tail + body + data_size

End Function
```

11. *Bibliography*

1. User Model for ISO-11172 Decoding Synchronization
2. ISO/IEC 13818-1 CD (mpeg-2 systems ctte)
3. Video CD Decoder Pack Parser Specifications

Address	'h167
Type	R

Bit #	Reset	Brief Description
4	0	not_en_scr_5
3	0	not_en_scr_4
2	0	not_en_scr_3
1	0	not_en_scr_2
0	0	not_en_scr_1

Address	'h168
Type	R/W

Bit #	Mode	Brief Description
5	R	DTS/DSMçount [0] (not detected by comparison with SCDcount)
4	R	SCD equal CD
3	R	dts_dsm_fifo_not_empty
2	R/W	read_dts_dsm_count_fifo
1	R/W	read_dts_dsm_value_fifo
0	W	lock

If *lock* is set, writting E8[2] will provide a read in DTS/DSMcount_fifo and the internal read signal is deconnected.
If *lock* is set, writting E8[1] will provide a read in DTS/DSMvalue_fifo and the internal read signal is deconnected.

See *test_cdunit601.wvs* (including counters to test DTS_association block) in STicoreBB database.

5	0	not_load_phl
2	0	not_en_flags

Address	'h165
Type	R

Bit #	Reset	Brief Description
3	0	start-code-duration
2	0	not_reset_flags
1	0	dec_phl
0	0	dec_pl

Address	'h166
Type	R

Bit #	Reset	Brief Description
7	0	not_en_DSM
6	0	not_en_dts_5
5	0	not_en_dts_4
4	0	not_en_dts_3
3	0	not_en_dts_2
2	0	not_en_dts_1
1	0	DTS_DSM_fifo_full
0	0	not_en_video

10. *Test Registers*

Address	'h160
Type	R/W/lockable

State-registers of the main state-machine.

Address	'h161
Type	R

packet length [15:8]

Address	'h162
Type	R

packet length [7:0]

Address	'h163
Type	R

packet-header-length[7:0]

Address	'h164
Type	R for [2:7] / W for [0]

Setting E4[0] will lock packet-length[15:0] , packet_header_length[7:0], pts_dts_flags[1:0], escr_flag, es_rate_flag and dsm_trick_mode_flag.

Bit #	Reset	Brief Description
7	0	not_load_msb_pl
6	0	not_load_lsb_pl

Mnemonic	VID_TS2
Address	'h4D
Type	R

VID_TS2 contents TS[15:8].

Mnemonic	VID_TS3
Address	'h4E
Type	R

VID_TS3 contents TS[23:16].

Mnemonic	VID_TS4
Address	'h4F
Type	R

VID_TS4 contents TS[31:24].

Type	R

PES_SC3 contents SCR[23:16].

Mnemonic	PES_SC4
Address	'h47
Type	R

PES_SC4 contents SCR[31:24].

9.5. TS value (33 bits)

Mnemonic	VID_TS5
Address	'h4B
Type	R

Bit #	Reset	Brief Description
3	0	SCR[32]
2	0	DSM_association_flag
1	0	TSA (DTS/PTS_association_flag)
0	0	ts[32]

TSA : Time-stamp associated.
This bit indicates if the picture now decoded has got a associated time stamp.
The DSM_association_flag indicates if the picture now decoded has got a associated DSM value.

Mnemonic	VID_TS1
Address	'h4C
Type	R

VID_TS1 contents TS[7:0].

Address	
Type	R

Bit #	Reset	Brief Description
7	0	new discarded packet
6	0	inconsistency error in pes level
5	0	new SCR
4	0	
3	0	Video-Core Interrupts
2	0	
1	0	
0	0	

9.4. SCR value (33 bits)

These registers contains the value of the SCR (mpeg-2 program-stream).

Mnemonic	PES_SC1
Address	'h44
Type	R

PES_SC1 contents SCR[7:0].

Mnemonic	PES_SC2
Address	'h45
Type	R

PES_SC2 contents SCR[15:8].

Mnemonic	PES_SC3
Address	'h46

9. Registers

9.1. Configuration of video parsing

Mnemonic	PES_CFG
Address	'h48
Type	R/W

Bit #	Reset	Brief Description
6	0	storeDTS_notPTS
5	0	input_format
4	0	ignore video_stream_id
3	0	Video_stream_ID [3:0]
2	0	
1	0	
0	0	

input_format : 0 => Compressed Data is a video elementary stream.
1 => Compressed Data is a system stream (mpeg-2 video pes stream or mpeg-2 program-stream)

If *input_format* remains reset, all the pes_parser block is reset.

9.2. DSM trick mode value and flag

Mnemonic	VID_DSM
Address	'h4A
Type	R

VID_DSM contains DSMvalue (8 bits)

9.3. Interrupt Status, Interrupt Mask and Status

Mnemonic	ITS - ITM - STA

6. Reset

Hard_reset resets all the configuration registers, specially the input-format bit.
Soft_reset has no impact on pes_parser.
Input-format bit can be called *not_reset_pes_parser*. When this bit is 0, it

- clears the state-machine (returns to IDLE).
- clears all the registers except the configuration.
- clears the pointer of DTS fifo and DSM fifo.

Pes-parser power-down mode : Forcing *Input-format* bit to 0, the pes_parser is "spleeping" that is to say all the data bypass the pes_parser.

7. Video Packet Synchronisation

he video packet synchronisation is verified directly at the PES layer by detection of the packet start codes and stream_id. If there is an error in the video decoding the video decoder will carry out the error concealment. This is detailed in section 8.

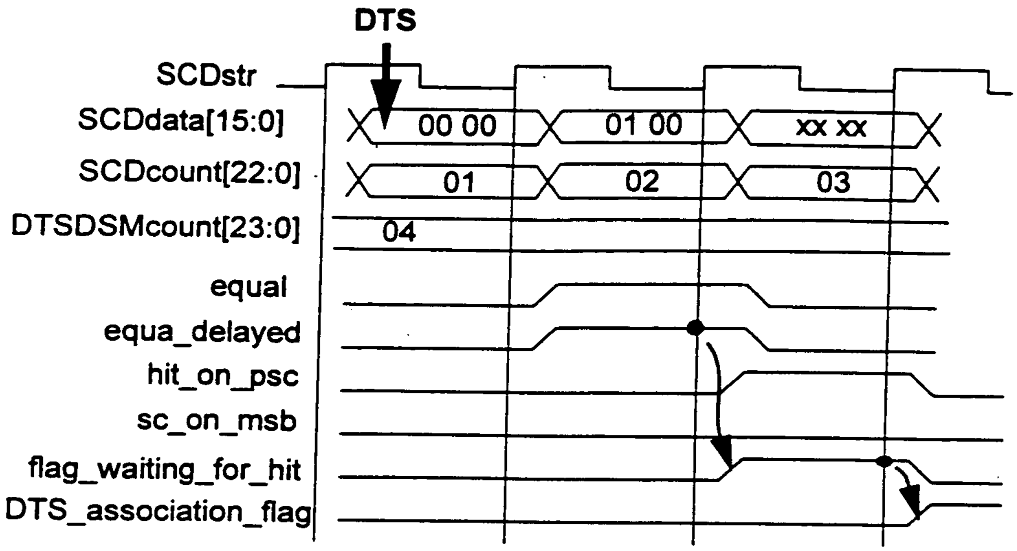
The parser assumes that the stream coming from the transport device is always byte aligned.

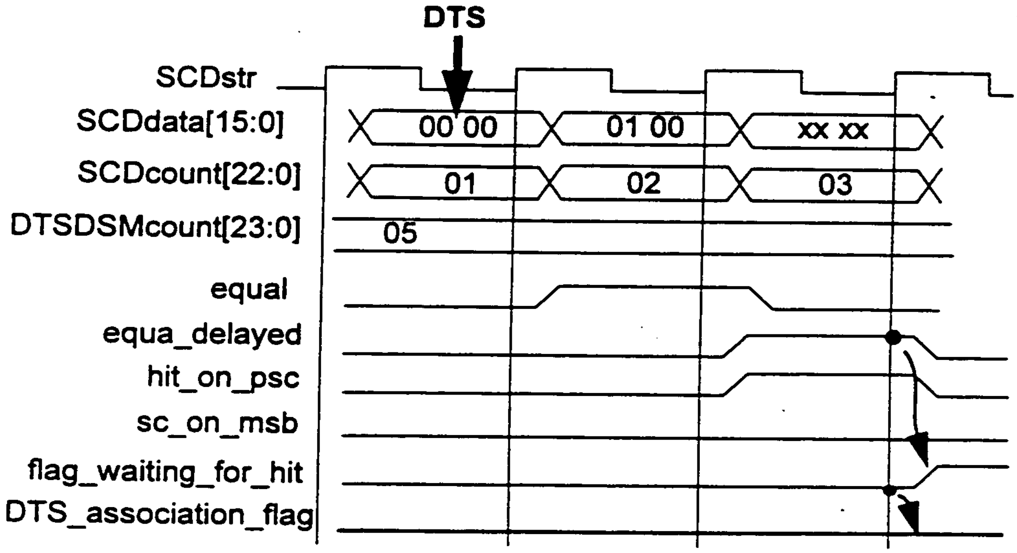
8. Errors

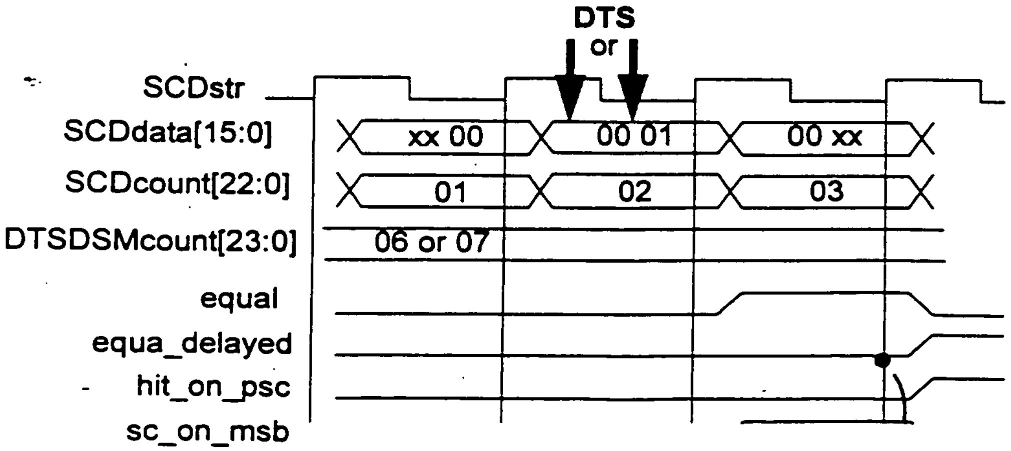
There will be no extra error protection at the PES level. The elementary stream decoders are very well protected against erroneous data.

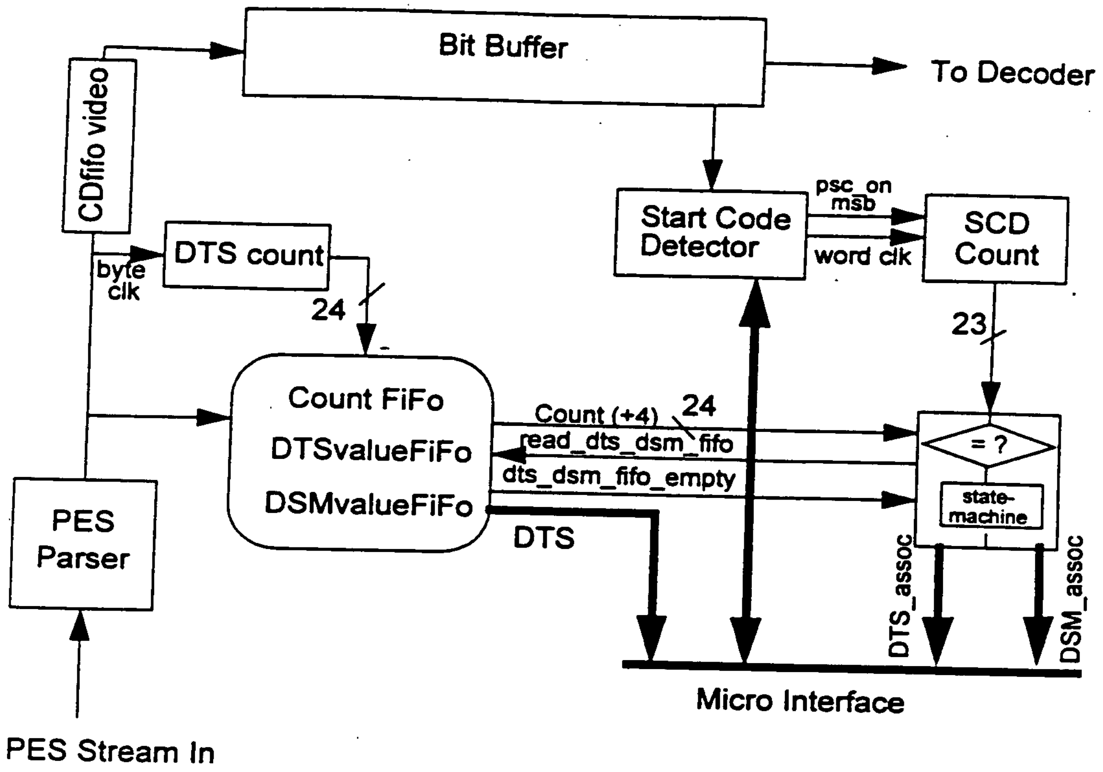
If an error occurs in video PES then an error start code is placed in the stream even if the error occurred in non-video (system) data. Error protection/concealment in the video decoder will be sed.

an error is detected in the audio stream then the stream should be stuffed with the correct number of missing bytes (1 packet = 188 bytes) in order to maintain audio frame synchronisation. If this is not done the audio decoder can loose synchronisation and depending on the value written in the SYNC_LOCK register.


DTS
SCDstr
SCDdata[15:0]
00 00
01 00
xx xx
SCDcount[22:0]
01
02
03
DTSDSMcount[23:0]
04
equal
equa_delayed
hit_on_psc
sc_on_msb
flag_waiting_for_hit
DTS_association_flag

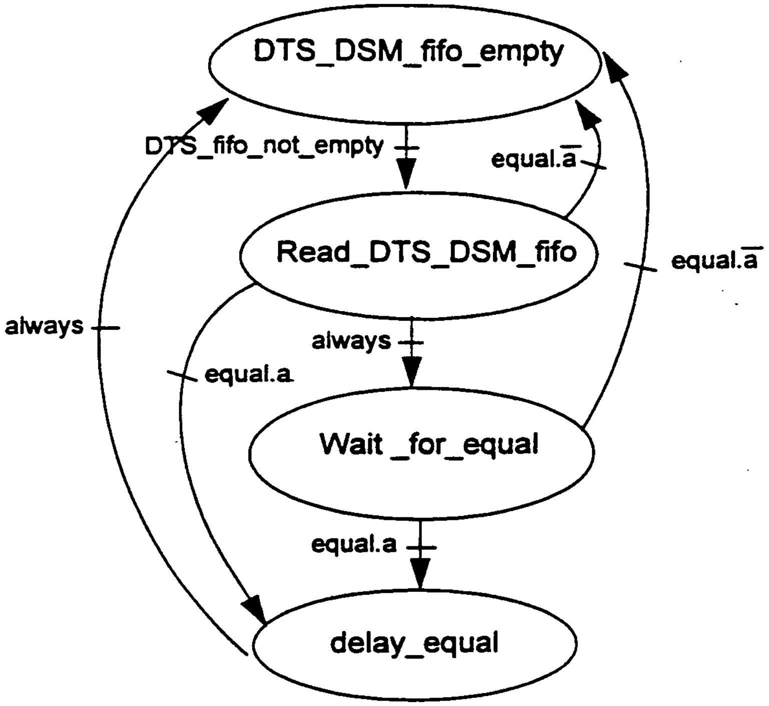

DTS
SCDstr
SCDdata[15:0]
00 00
01 00
xx xx
SCDcount[22:0]
01
02
03
DTSDSMcount[23:0]
05
equal
equa_delayed
hit_on_psc
sc_on_msb
flag_waiting_for_hit
DTS_association_flag


DTS
or
SCDstr
SCDdata[15:0]
xx 00
00 01
00 xx
SCDcount[22:0]
01
02
03
DTSDSMcount[23:0]
06 or 07
equal
equa_delayed
hit_on_psc
sc_on_msb



DTS and DSM Synchronisation Model Fig 2.

The maximum size of the bit buffer is the memory size (128 Mbits). The counter must be able to count all words from the memory so we need a 24 bits counter.
The count of the data read by the start code detector will be *SCD-count*.
DTS/DSM-count is a byte count, and SCD-count is a word count, because 16-bits data are read by the start code detector. The start code detector must provide the signal "*hit_on_PSC*". DTS/DSM-count[0] (DTS/DSMcount parity) and the signal sc_on_msb are used to delay equal when DTS take place in a critical position of the bitstream (and the same for DSM) :



a : (DTS[0]=1 or sc_on_msb = 1)

Soft operation : The micro can check *DTS_association_flag* and *DSM_association_flag* on a occuring vsync for instance, and if set can read the DTS_value and DSM_value and compare it to the STC_counter. A skip or wait instruction can be launch on the next picture, depending on the result.

5. TIme Stamp Association

5.1. Video

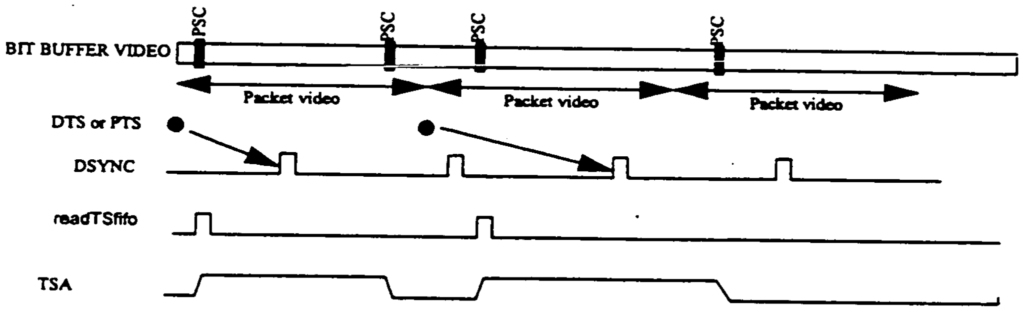
The DTS is extracted from the PES header along and held in a register (DTS_value). When the DSync interrupt is received and DTSflag is high the image currently being decoded has a time stamp (DTS_value) associated with it. The same mechanism can be used to perform the association of the DSM trick mode which is also extracted from the PES header.

VSync is used to latch the current value of the 90 KHz clock so this is available for every image.

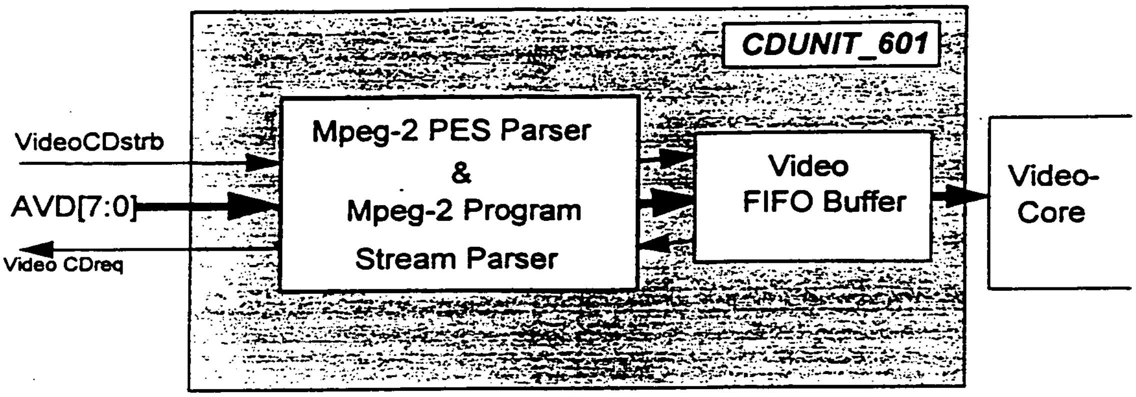
5.2. Detailed Operation

At the input to the device the parser strips out the SCR from the PACK HEADER, the DTS [1] and the DSM trick mode from the PES HEADER. The SCR is placed in a register which can be read by the external microprocessor. The DTS is placed in the DTS fifo (refer to diagram below) and the DSM into the DSM fifo with the PTS/DTS flag and the DSM trick mode flag. The byte count of the data read by the start code detector is compared with the current value of DTS at the output of the fifo DTS/DSM count and a flag is raised (*flag_waiting_for_hit*) when the byte count from the start code detector becomes equal to the current byte count at the output of the DTS/DSM fifo count. When a Picture_Start_Code is found by the start-code-detector, *DTS_association_flag* and DSM_association_flag take the value of *flag_waiting_for_hit* if the corresponding flag stored in the fifo tells that this field was present in the bit stream. Thus, *DTS_association_flag (TSA)* and *DSM_association_flag* are set between the associated PSC and the next one.
Each time one of the two fields DTS/PTS and DSM trick mode are present, both will be stored in their fifo and a flag will indicate to the user which one is correct.

A bit will allow to select DTS or PTS storage.



In order to prevent if DTS_DSM fifos are empty, a simple state_machine is used :



PES Parser Global Architecture Fig 1.

3. *PES Parser Changes Relative to the 3520A Parser*

The following changes have been made :

- The history fifo is removed : every non video data are discarded.
- DTS/PTS fifo : when the bit buffer is big, the last value written in the DTS/PTS value fifo is over-written so this has been modified.
- DSM trick mode : this field has to be stored like the DTS/PTS value with an association flag.
- Audio data are also discarded.
- Only MPEG2 program or pes streams and no more MPEG1 system streams.
- The ESCR value (PES header) is not stored. We only store the SCR value (pack header).

4. *Functionnality*

When the *input-format* bit is set, a system stream is expected.

.The pes-header length is still used to detect the end of pes header.
Important : The length of pes-video cannot be used because it can be zero (means undefined). In this case, the end of a pes-video (and a pes-video discreminated by the stream-id) will be detected by a new pes-start-code.

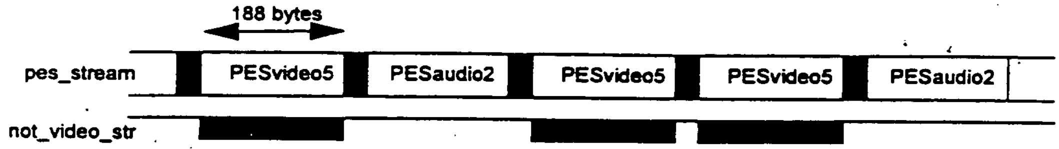
1. Introduction

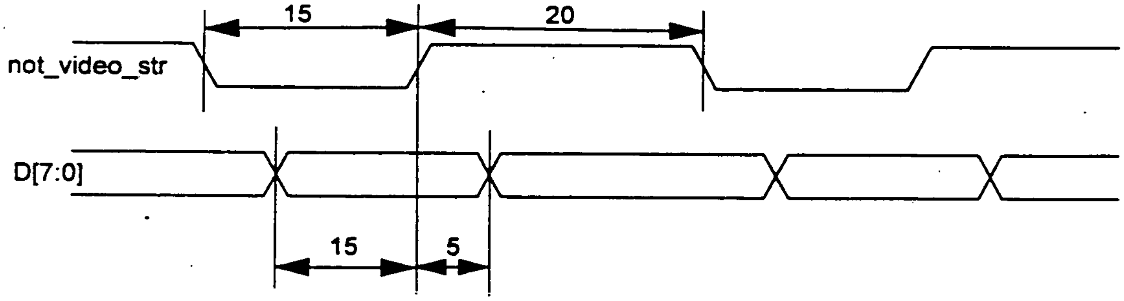
The block is situated between the MCU interface and the compressed data fifos of the video core.

Bit rate :	100 Mbits/sec (Max burst)
Formats Accepted:	Packetized PES (MPEG-2) Audio & Video (iso 13818)

2. Functional Description

The HDTV accepts PES in the same way as the pure audio or video streams were accepted in the STi3520A. It accepts also program stream. The difference is that video data are transmitted to the bit buffer and audio data are discarded.
When the device is configured to accept PES then the strobe and request signals will refer to packetised Video PES.





not_video_str **will be the clock** of the pes_parser block.

HD MPEG VIDEO DECODER

APPENDIX U

PES PARSER

Thomson Consumer Electronics, Inc.
Indianapolis, Indiana, USA

Revision No. 2.0

THESE DRAWINGS AND SPECIFICATIONS ARE THE PROPERTY OF THOMSON CONSUMER ELECTRONICS, INC. AND SHALL NOT BE REPRODUCED OR COPIED OR USED AS THE BASIS FOR THE MANUFACTURE OR SALE OF APPARATUS OR DEVICES WITHOUT WRITTEN PERMISSION.

TCE PROPRIETARY AND CONFIDENTIAL

In addition to the sate machine, a counter is provided for providing control signals that change the state of the state machine. This counter takes the DFS information at the falling edge of the pictReset pulse.

A macroblock starts every 3 BSync. Bsync is active on the same cycle of the first run-level or EOB coefficient of a block:

The macroblock counter is used to recognize the end of the picture.
The count is reset on pictReset.

In case of error concealment, an ercReset is generated. This reset resets the state machine but not the macroblock counter. Its action on the macroblock counter is to disable the next count (i.e. the first macroblock after an ercReset is not counted).

3.3. Input interface

The input interface is not strictly a pipeline input : when the pipe controller receive one information (run or EOB), it generally means several data.

- run = n means n+1 data are generated in the pipeline
- EOB means number of data have to be completed to 64 (one block) since the previous EOB. If a block is only composed of an EOB, then 64 data are generated in the pipeline.

This means that when an information is received in the controller, the vldReq signals falls until correct number hof data are generated.

3.4. Clock generation and output stage

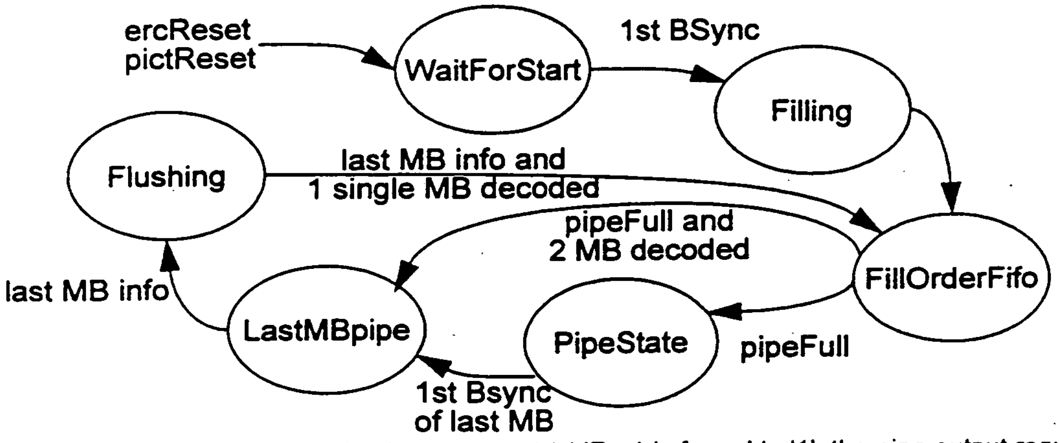
Clock of the pipeline (pclk) is only generated when the pipeline runs (i.e. when filling and receiving data from the VLD or when outack is high, i.e. when data are output of the pipeline).

- clkInRLD : clock signal, active when a new run-level or EOB information has to be strobed in the RLD.
- notZero : low when zero runs have to be output from the RLD.
- reorder_in : signal is asserted when data coming out of the idct are valid and can be written to the reordering fifos.
- reorder_out : signal is asserted to allow the reordering fifos to be read.

3. *Functionality*

3.1. *State machine*

The first control part is a state machine representing the different states of the pipeline.



Basically, when in Filling state, the pipe input request (vldReq) is forced to '1', the pipe output request (outreq) is forced to '0'. The reorder_in signal also is forced to '0'.

In FillOrderFifo, vldReq is forced to '1', outreq to '0' and reorder_in is depending on vldAck : reordering fifos are filled.

In PipeState, the pipe is running as a fix length pipeline : one input generates one output, so the outreq, vldReq, reorder_in and reoder_out signals are linked to vldAck and outack.

In LastMBpipe, the function is the same, but this state prepare the Flushing state.

in Flushing state, the input request (vldReq) is forced to '0' and the output request to '1'.

A pixel counter (clock edge counter) is associated to the state machine to determine the fullness of the pipe and of the reordering fifos, the number of cycle of LastMBpipe state in the different cases.

The latency of the RLD-IQ-IDCT chain is 159 data. The latency of the reordering fifos is 128 data (fixed number). So the total of the pipe is 287 data.

One macroblock is 192 data (only half of a total macroblock enters one pipe).

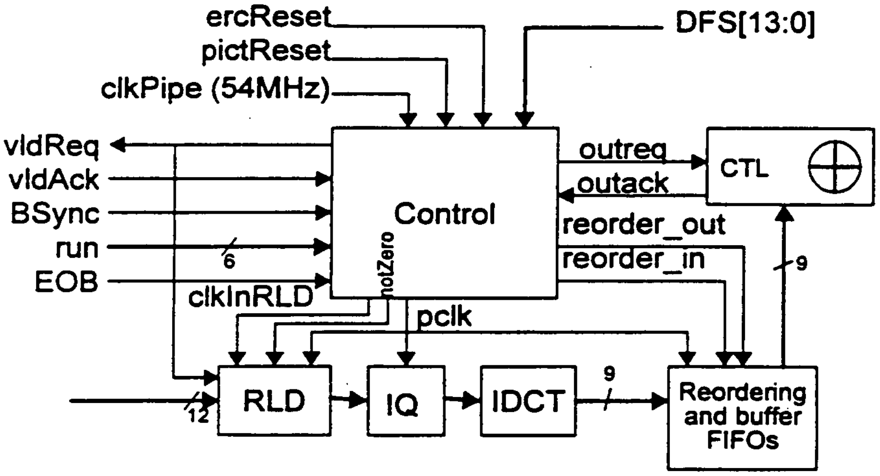
It may be possible, due to error concealment, that only one macroblock or two are decoded in the pipeline. That is why special transition are possible in the state machine.

3.2. *Macroblock counter*

1. *Introdaction*

This document describes the pipe controller block. This controller is present in each pipe of the IC. Its function is to generate the clock of the associated pipe, to receive and assert the control signals of this pipe.
It is also used to control the filling of the pipe at the beginning of a picture or when starting error concealment and to control the flushing of the pipe at the end of a picture.

2. *Description of interfaces*



2.1. *General interface*

- clkPipe : pipe clock coming from the clock generator (typically 54 MHz)
- pictReset : synchronous reset signal asserted before every start of picture
- ercReset : synchronous reset signal asserted to reset the pipeline before error concealment
- DFS[13:0] : decoded frame size, number of macroblock of the decoded picture.

2.2. *VLD interface*

- Run vector : 6 bits run coefficient
- EOB signal : end of block signal
- BSync signal : block synchronization, correspond to a start of a block
- vldReq : request signal to the VLD
- vldAck : validation signal, when low all other signals have no meaning, when high they can be sampled

2.3. *Output interface*

- outreq : request adder for outputting data of the pipe
- outack : when high, data are output of the pipe.

2.4. *Pipe interface*

- pclk : clock of the pipeline. This clock is coming from the clock generator.

HD MPEG VIDEO DECODER

APPENDIX T

PIPE CONTROLLER DESIGN

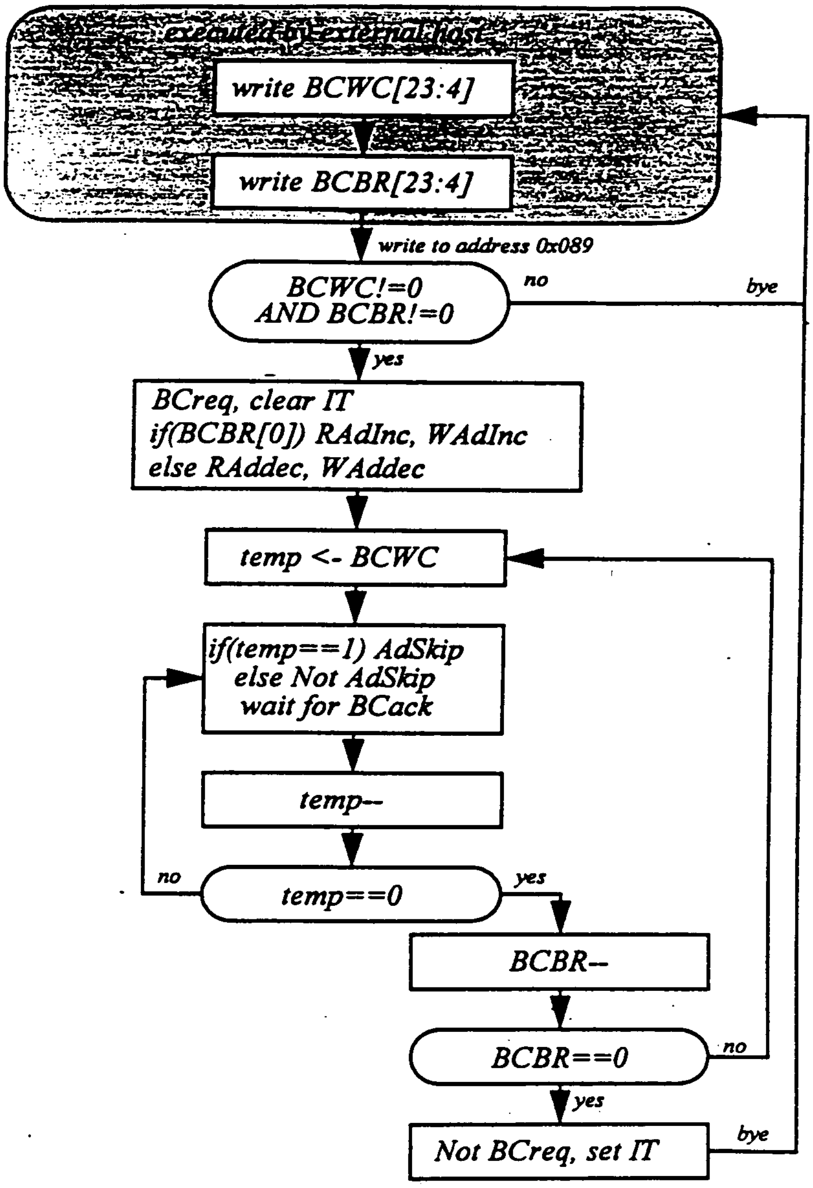
Thomson Consumer Electronics, Inc.
Indianapolis, Indiana, USA

Revision No. 1.0

THESE DRAWINGS AND SPECIFICATIONS ARE THE PROPERTY OF THOMSON CONSUMER ELECTRONICS, INC. AND SHALL NOT BE REPRODUCED OR COPIED OR USED AS THE BASIS FOR THE MANUFACTURE OR SALE OF APPARATUS OR DEVICES WITHOUT WRITTEN PERMISSION.

TCE PROPRIETARY AND CONFIDENTIAL

ming error.
If none of BCBR and BCWC are zero, the block copy mode proceed execution and an IT is generated when the process is achieved.



Except for the last mode, the read process is started as soon as the register HRC is written.
Automatic read mode starts as soon as the HRC register is written. The difference is that this mode progress by itself each time register file 15 (@0x0BF) is read.

2.3. the host computer write into the memory.

Four possibilities are available according to HRC register value (@0x008) set by the host:

- HRC[6]=simple write. The register file value drives the memory bus.
  To LMC: LMWreq.
  From LMC: LMWack
- HRC[4]=write and increment. The register file value drives the memory bus. Then the write pointer increments.
  To LMC: LMWreq, WAdInc.
  From LMC: LMWack
- HRC[2]=write and decrement. The register file value drives the memory bus. Then the write pointer decrements.
  To LMC: LMWreq, WAddec.
  From LMC: LMWack
- HRC[0]=automatic write. An automatic transfer of the register file value to the memory bus is performed when the host computer write the content of register file number 15 (@0x0BF). Then the write pointer increments.
  To LMC: LMWreq, WAdInc.
  From LMC: LMWack

Except for the last mode, the write process is started as soon as the register HRC is written. Write latch must already been settled.
Automatic write mode starts as soon as the HRC register is written. The difference is that this mode progress by itself each time register file 15 (@0x0BF) is written.
It is important to understand that automatic write mode could generates unwanted access to memory. Let's say that a simple write mode follows an automatic mode. Before writing HRC register, data must be prepared into the register file. If register file 15 is set before others, the automatic mode is started (since not inhibited), and the 128 bits data is written in memory at the current address, followed by an address increment.

2.4. Inhibition.

Inhibition is done by inactivating BCreq, LMRreq and LMWreq.

2.5. the host computer read the status of the block.

Informations provided to the IC status bus are the following :

- read is not ready.
- write is not ready.
- Block copy done (i.e. block copy idle, IT in next page diagramm).

2.6. the host computer initiates a block copy.

The block copy mode is started when high byte of register BCBR[23:16] (@0x08B) is written. This mode freezes at the end of execution after reception of the last read/write acknowledge (SIGNAL BCack, generated from LMWack). While the block is operating, BCreq signal remains always active.
Bit 0 of BCBR defines the forward mode, which correspond to address increment. If not set, address decrement.
Bit 1 of BCBR , if set, programs a word copy : the read pointer address is not incremented and the same word is written all over the destination area.
If BCBR and BCWC registers have the zero value, which is not meaningful, no IT is generated, no block copy is performed. It is the responsibility of the host computer to time out the block copy process to detect this program-

@0x088 is a dummy address, @0x089 for BCBR[23:16]
@0x08A for BCBR[15:8], @0x08B for BCBR[7:0].

- register BCWC[23:4] (block copy word count): set how many words are processed before an address skipping occurs in block copy mode.
  @0x08C is a dummy address, @0x08D for BCWC[23:16]
  @0x08E for BCWC[15:8], @0x08F for BCWC[7:0].
- The status of operation is transferred to the status bus of the IC, and so available wether as status or interruption.

2.1. Address of register file.

The name given to register file is MAF[127:0]. Address of individual bytes for the host processor are:

- @0x0B0 for MAF[127:120].
- @0x0B1 for MAF[119:112].
- @0x0B2 for MAF[111:104].
- @0x0B3 for MAF[103:96].
- @0x0B4 for MAF[95:88].
- @0x0B5 for MAF[87:80].
- @0x0B6 for MAF[79:72].
- @0x0B7 for MAF[71:64].
- @0x0B8 for MAF[63:56].
- @0x0B9 for MAF[55:48].
- @0x0BA for MAF[47:40].
- @0x0BB for MAF[39:32].
- @0x0BC for MAF[31:24].
- @0x0BD for MAF[23:16].
- @0x0BE for MAF[15:8].
- @0x0BF for MAF[7:0].

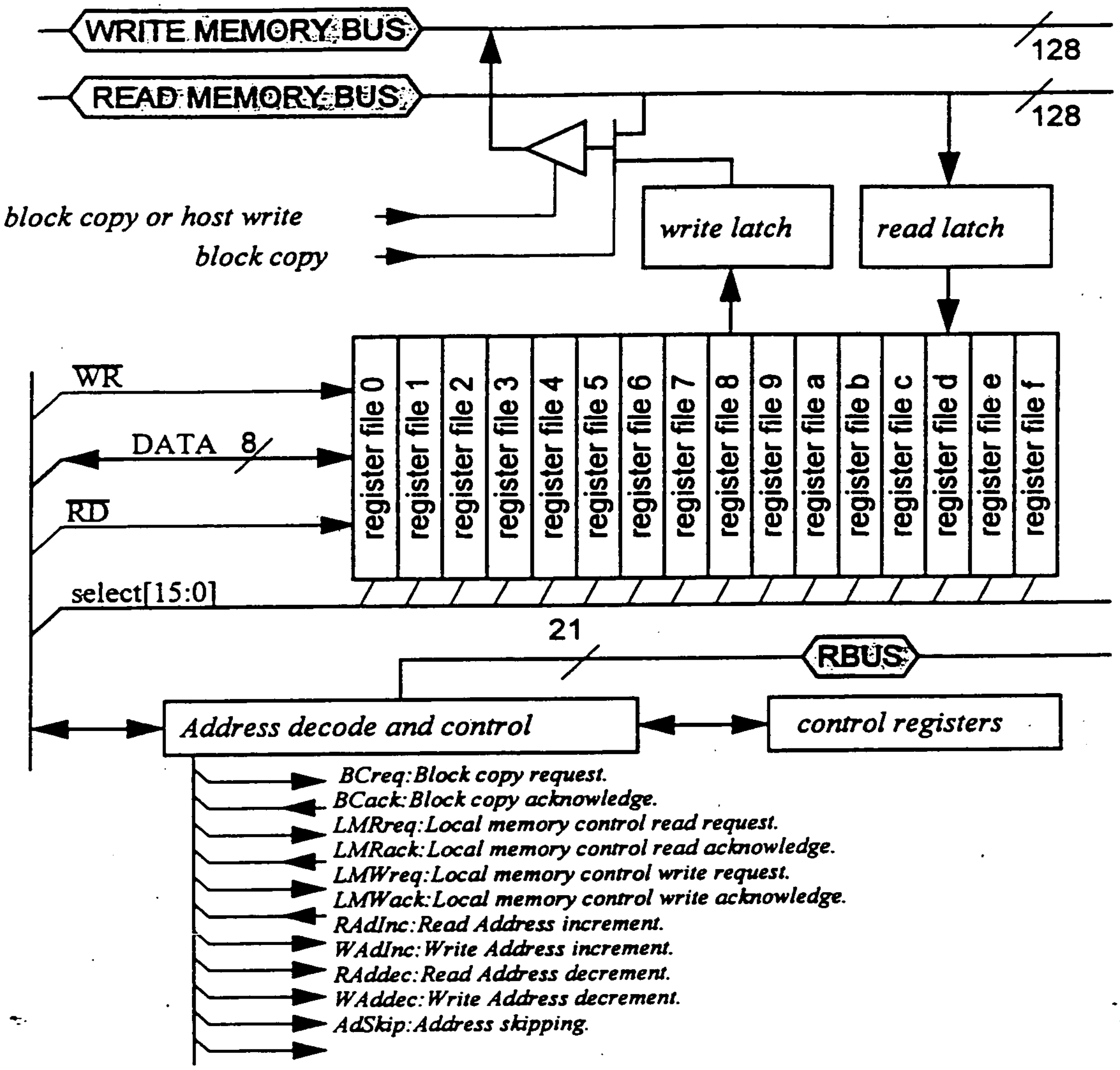
2.2. the host computer read from the memory.

Four possibilities are available according to HRC register value (@0x008) set by the host:

- HRC[7]=simple read. The memory bus value is stored into the register file.
  To LMC: LMRreq.
  From LMC: LMRack
- HRC[5]=read and increment. The memory bus value is stored into the register file. Then the read pointer increments.
  To LMC: LMRreq, RAdInc.
  From LMC: LMRack
- HRC[3]=read and decrement. The memory bus value is stored into the register file. Then the read pointer decrements.
  To LMC: LMRreq, RAddec.
  From LMC: LMRack
- HRC[1]=automatic read. An automatic storage of memory bus value is performed into the register file when the host computer read the content of register file number 15 (@0x0BF). Then the read pointer increments.
  To LMC: LMRreq, RAdInc.
  From LMC: LMRack

1. Overview.

The local memory interface is the place where data exchange occurs between external memory and host bus (RBUS). One read and one write transfer latches are used to interface the read or write memory bus.
In block copy mode a transfer of data is performed between one memory area to another. In this particular case a direct path from read to write memory bus is used.



2. operating modes.

Three registers are set to define the operating mode of this block:

- register HRC[7:0] (host register control, @*0x008*): defines the operating mode of this block.
- register BCBR[23:4] (block copy block repeat): set the number of loops to execute in block copy mode.

HD MPEG VIDEO DECODER

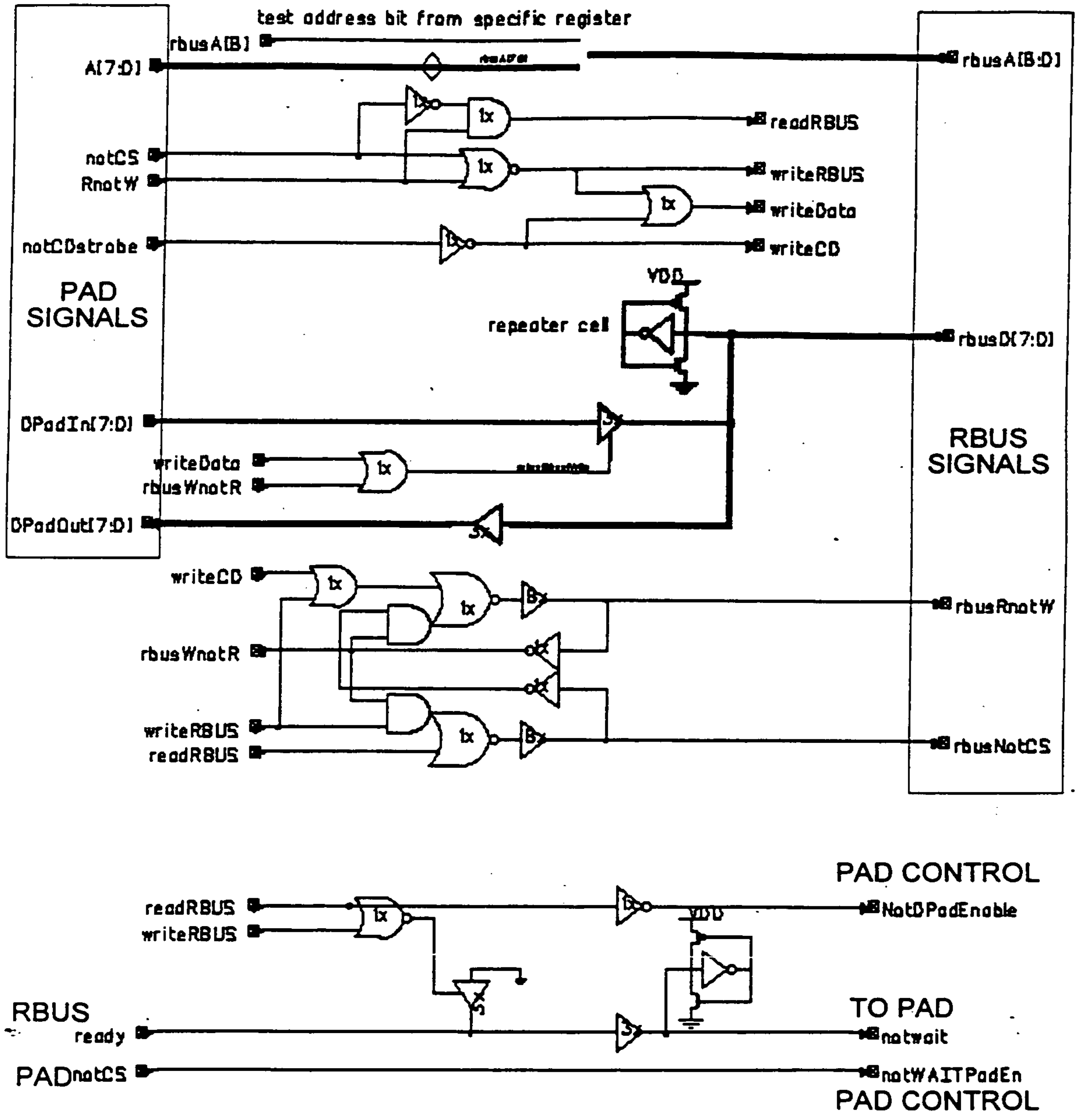
APPENDIX S

LOCAL MEMORY INTERFACE

Thomson Consumer Electronics, Inc.
Indianapolis, Indiana, USA

Revision No. 1.1

TCE PROPRIETARY AND CONFIDENTIAL

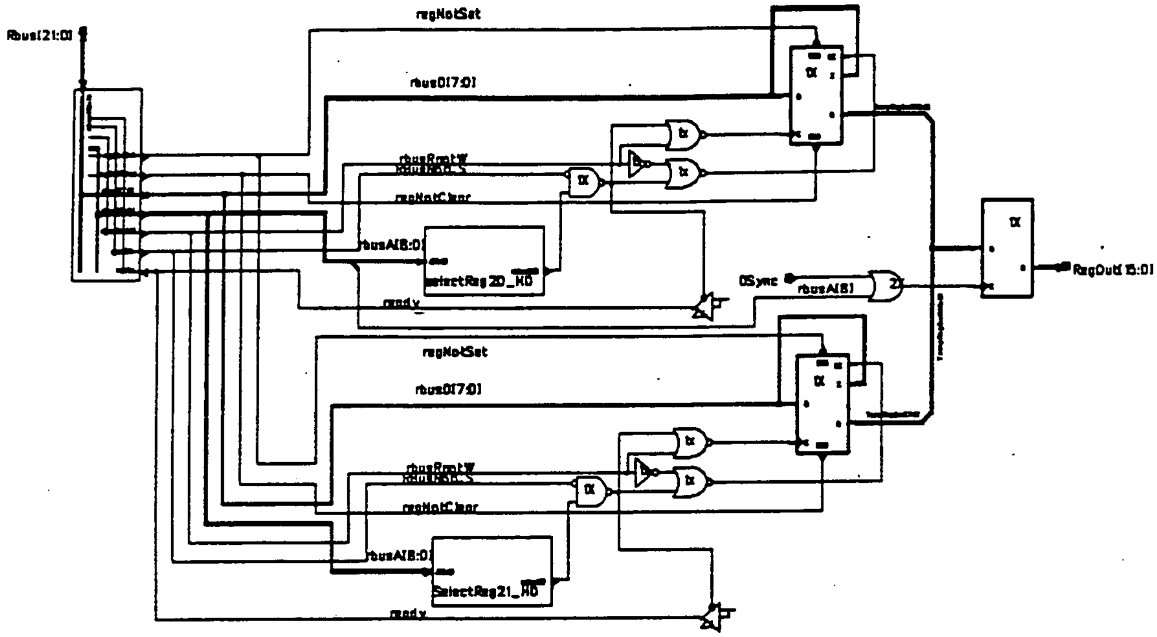


7. *Specific register description*

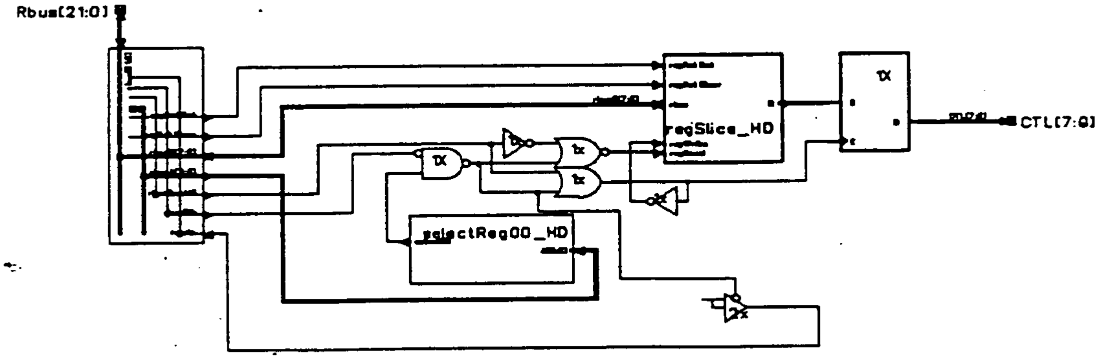
An specific topic concerning application registers is that a lot of them need double buffering, because their action in the chip can not occur at any time. The double-buffering consist in putting a latch on a specific signal just after the register.

For the test purpose, it is interesting to bypass the double buffering. This is done by opening the latch with the test address bit rbusA[8].

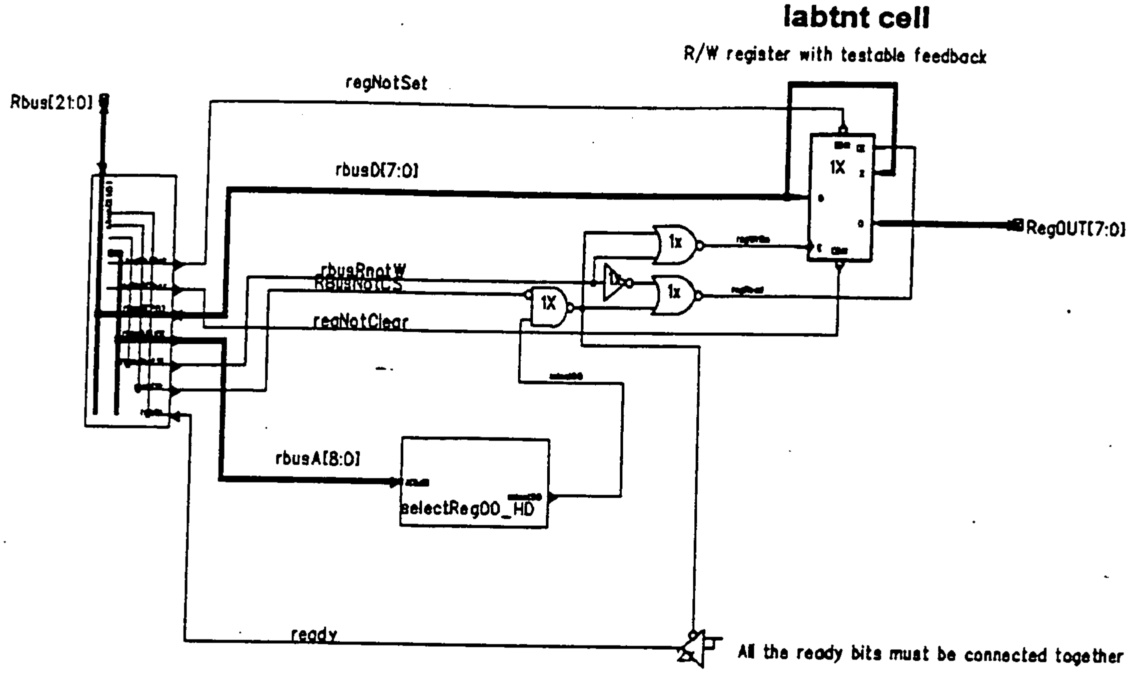
Example of a 16 bits register double-buffered on DSync



Some registers need to be edge triggered on rising edge of external nCS and not only latches. In this case, a latch is added after the register cell. Example : CTL reg



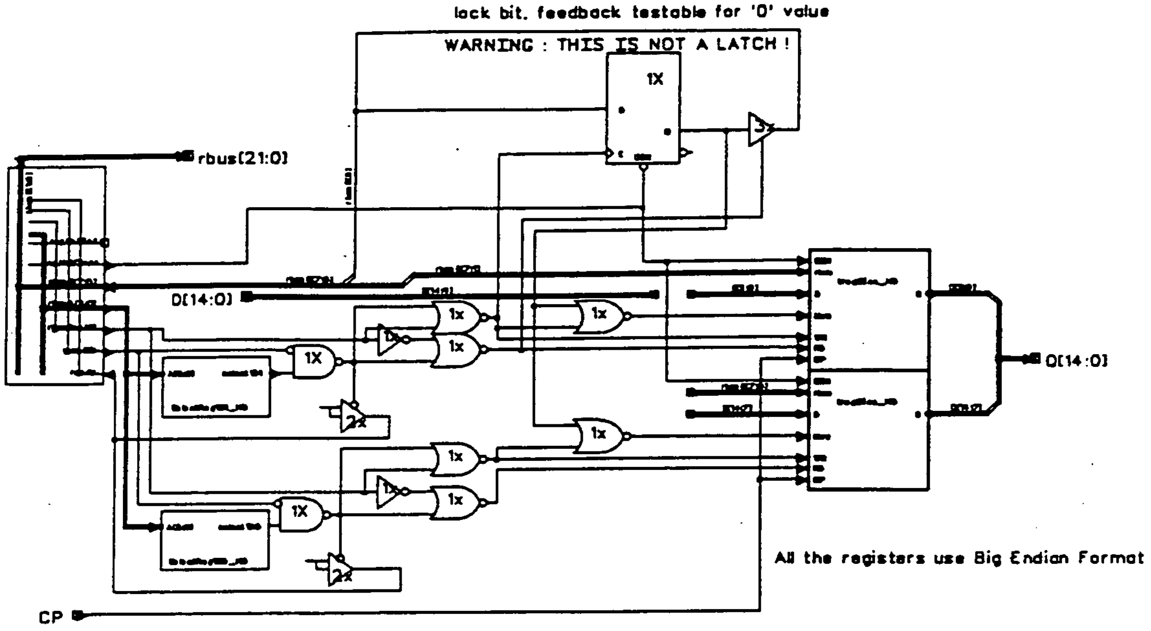
Application register schematic



5. *Test register description*

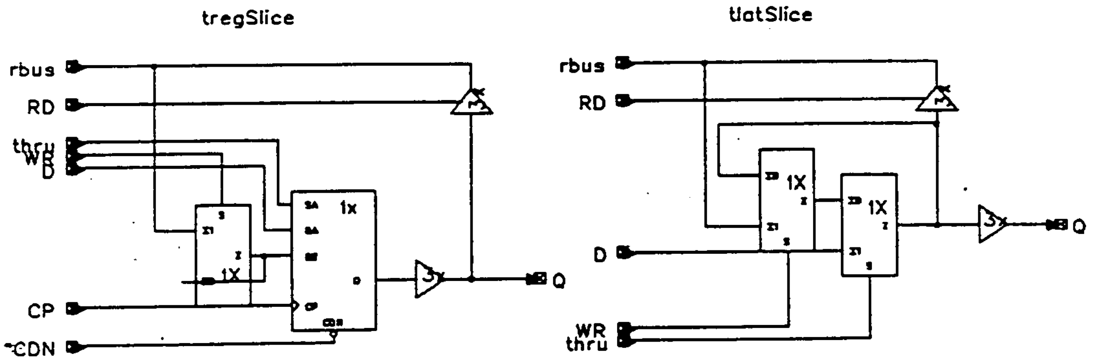
There are two kind of test registers : test latches or test flip-flops.
Test registers are inserted at some points in the design to get a direct parallel input into blocks and/or to get a direct parallel observability.
Test registers contain a lock bit. When this bit is reset, the register is transparent. When it is set, the register output is locked to the value written into it.
The test latches, when transparent, correspond to a buffer, with a specific timing delay. They can be inserted thru a wire.
The test flip-flops, when transparent, correspond to a flip-flop. They can be inserted where the design already contains a flip-flop.
Some additional bit can be added to the test registers, that are register only bits (they do not correspond to a path of the design). They are used to control specific parts to help the test generation.
To have a greater amount of parallelism, test registers can be grouped, containing only one lock bit for all paths.
The lock bit is static testable for the value '0' (non locked), with RegNotClear signal.

Schematic example of a test register, with 15 bits path and one lock bit


lock bit. feedback testable for '0' value
WARNING : THIS IS NOT A LATCH !
1X
rbus[21:0]
D[14:0]
Q[14:0]
1x
1x
1x
1X
2x
1x
1x
1x
1X
2x
3x
CP
All the registers use Big Endian Format

The transparent path for the functional use is for D to Q. This test register, which is a test flip-flop, is equivalent to a D flip-flop with the CP clock.
To get a test latch, the tregSlice block is replaced by a tlatslice, CP and CDN are no more used.

Schematic of the tregSlice and tlatSlice



Buffering of Q can be adapted.

6. *Microcontroller interface description*

Following is the schematic description of the microcontroller interface.

1. Introduction

This document describes the internal microcontroller interface of the HD MPEG IC. Registers are distributed in the chip and connected to an internal bus. The purpose of this document is to specify the registers, the internal bus and the interface between internal and external bus.

2. External bus description

The external interface contains 20 signals :

•A[7:0]	input address bus
•D[7:0]	bidirectional data bus
•R/nW	input Read/notWrite signal
•nCS	chip select input negative strobe (access to register)
•nSTRB	video compressed data input negative strobe
•nWAIT	output not wait signal

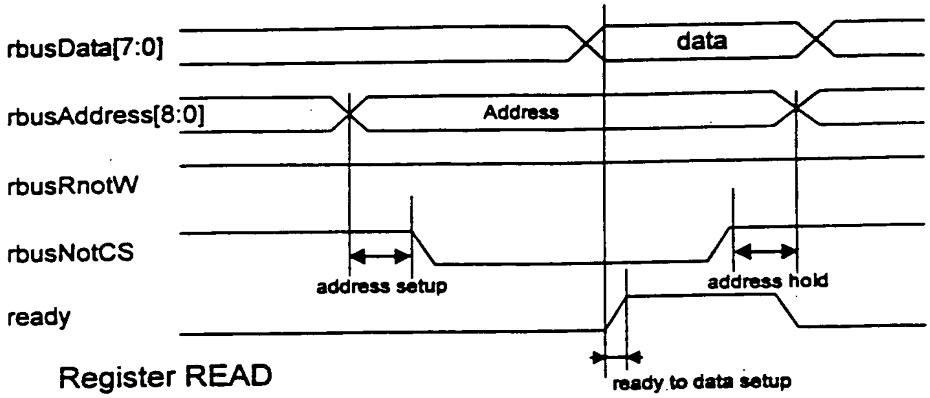
3. Internal bus description

The internal 22-bits register bus (rbus [21:0]) is split into :

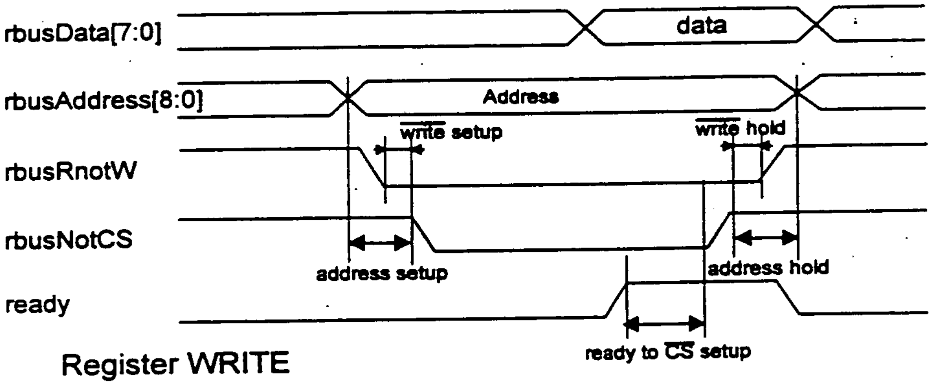
•rbus[0] = rbusRnotW	Readnot Write signal, from micro interface
•rbus[1] = rbusNotCS	negative strobe to validate the read or write, from micro interface
•rbus[10:2]= rbusAddress[8:0]	address bus, from micro interface
•rbus[18:11]= rbusData[7:0]	data bus, bidirectional
•rbus[19] = ready	ready signal, bidirectional, to generate nWAIT
•rbus[20] = regNotClear	signal use for testing, form micro interface
•rbus[21] = regNotSet	signal use for testing, from micro interface

RegNotClear and RegNotSet are asynchronous signals used for static register testing. RegNotClear clears all registers, RegNotSet sets all non test registers. RegNotClear must be at least hardReset inverted.

Internal bus access diagramms



Register READ



Register WRITE

4. *Application register description*

The read pulse of a register is obtained from rbusNotCS pulse thru address and rbusRnotW decoding.
The write pulse of a register is obtained from rbusNotCS pulse thru address and rbusRnotW decoding.
The ready signal is low when no register is accessed and rises from the accessed register during an access. The load on the ready net is equivalent to the maximum load on the rbusData bus, to emulate the worst case data transfer, from the register to the microcontroller interface when reading, or from the microcontroller to the register when writing. Drive strength of registers and microcontroller to the rbus are adjusted and standardized.

HD MPEG VIDEO DECODER

APPENDIX R

MICROCOMPUTER INTERFACE AND R-BUS

Thomson Consumer Electronics, Inc.
Indianapolis, Indiana, USA

Revision No. 2.2

THESE DRAWINGS AND SPECIFICATIONS ARE THE PROPERTY OF THOMSON CONSUMER ELECTRONICS, INC. AND SHALL NOT BE REPRODUCED OR COPIED OR USED AS THE BASIS FOR THE MANUFACTURE OR SALE OF APPARATUS OR DEVICES WITHOUT WRITTEN PERMISSION.

TCE PROPRIETARY AND CONFIDENTIAL

```
'Function macro: pred_bc_clks(    b_width, latency, stu_clks, mbs_per_ln  )
'
'This function computes the number of clock cycles required per H reset
'to read in 2 (best case) predictors from memory (forward only or backward
'only), and write each macroblock computed in that interval.  Each predictor
'is, by design, accessed from opposite banks of SDRAM.  Variables b_width
'and latency denote the bus width and CAS latency of the target SDRAM.
'Variable stu_clks represents the number of memory cycles required to access
'one storage unit in memory.  Variable mbs_per_ln represents the number of
'macroblocks processed per H reset

Function pred_bc_clks(b_width, latency, stu_clks, mbs_per_ln)

    clk_cnt = 0

    Select Case b_width
    Case 64
        'A storage unit is one 8 by 8 field block.  Six such st. units are
        'accessed to form one predictor.  Each predictor, given the memory
        'organization, can have at most one page crossing (in the upper 3
        'or lower 3 blocks but not both at the same time)

        'First macroblock in scan line ...
        'First field upper half ...
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 1, 1, 0, 1, 1, 1)
        '... Second field upper half ...
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 1, 1, 1, 1, 1, 0)
        '... First field lower half ...
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 1, 1, 1, 0)
        '... Second field lower half ...
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 1, 1, 1, 0)

        'Writing 4 storage units to memory:
        'Writes are not large enough to hide overhead
        'First writes in each field ...
        clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 1, 1, 1, 1)
        clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 0, 1, 1, 0)
        'Second writes in each field with page crossing (need precharge)
        clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 0, 1, 1, 0)
        clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 0, 1, 1, 0)

        'Intermediete macroblocks in scan line ....
        For i = 1 To (mbs_per_ln - 2) Step 1
            'First field upper half ...
            clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 1, 1, 0, 1, 1, 1)
            '... Second field upper half ...
            clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 1, 1, 1, 1, 1, 0)
            '... First field lower half ...
            clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 1, 1, 1, 0)
            '... Second field lower half ...
            clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 1, 1, 1, 0)

            'Writing 4 storage units to memory:
            'First writes in each field ...
            clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 1, 1, 1, 1)
            clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 0, 1, 1, 0)
            'Second writes in each field without page crossing (no precharge)
            clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 0, 0, 1, 0, 0)
            clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 0, 0, 1, 0, 0)
        Next i
        'Last macroblock in scan line
        'First field upper half ...
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 1, 1, 0, 1, 1, 1)
        '... Second field upper half ...
        clk_cnt = clk_cnt + read_clks((3 * stu_clks)., latency, 1, 1, 1, 1, 1, 0)
        clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 0, 0, 1, 0)
        '... Second field lower half ...
        clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 0, 0, 1, 0)
        '... and similarly for the other two predictors:
        clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 0, 0, 1, 0)
        clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 0, 0, 1, 0)
        clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 0, 0, 1, 0)
        clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 0, 1, 1, 0)
        'Writing 2 storage units to memory:
        clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 0, 1, 1, 1)
        clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 0, 0, 1, 0)
    End Select

    pred_wc_clks = clk_cnt

End Function
```

```
'organization, can have at most one page crossing (in the upper 2 units
'or the lower 2 units but not both at the same time)
'Note:  Tail overhead gives a more meaningful number of overhead cycles
'       This is not representative of what is assumed to be happening
'First macroblock in scan line ...
'First field upper half ...
clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 0, 1, 1, 1)
'... Second field upper half ...
clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 1, 0, 1, 0)
'... First field lower half ...
clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 1, 0, 1, 0)
'... Second field lower half ...
clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 1, 0, 1, 0)
'... and similarly for the other two predictors:
clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 1, 0, 1, 0)
clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 1, 0, 1, 0)
clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 1, 0, 1, 0)
clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 1, 0, 1, 0)
'Writing 2 storage units to memory:
'Writes are not large enough to hide overhead
clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 1, 1, 1, 1)
clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 0, 1, 1, 0)

'Intermediete macroblock in scan line ...
For i = 1 To (mbs_per_ln - 2) Step 1
    'First field upper half ...
    clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 0, 1, 1, 1)
    '... Second field upper half ...
    clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 1, 0, 1, 0)
    '... First field lower half ...
    clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 1, 0, 1, 0)
    '... Second field lower half ...
    clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 1, 0, 1, 0)
    '... and similarly for the other two predictors:
    clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 1, 0, 1, 0)
    clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 1, 0, 1, 0)
    clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 1, 0, 1, 0)
    clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 1, 0, 1, 0)
    'Writing 2 storage units to memory:
    clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 1, 1, 1, 1)
    clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 0, 1, 1, 0)
Next i

'And the last macroblock
'First field upper half ...
clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 0, 1, 1, 1)
```

```
    For i = 1 To (mbs_per_ln / 2) Step 1
        'First field upper half ...
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 1, 1, 1, 1, 1, 1)
        '... Second field upper half ...
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 1, 1, 1, 1, 1, 0)
        '... First field lower half ...
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 1, 1, 1, 0)
        '... Second field lower half ...
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 1, 1, 1, 0)
        '... and similarly for the other two predictors:
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 1, 1, 1, 1, 1, 0)
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 1, 1, 1, 1, 1, 0)
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 1, 1, 1, 0)
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 1, 1, 1, 0)
        'Writing 4 storage units to memory:
        'First writes in each field ...
        clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 1, 1, 1, 1)
        clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 0, 1, 1, 0)
        'Second writes in each field without page crossing (no precharge)
        clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 0, 0, 1, 0, 0)
        clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 0, 0, 1, 0, 0)
    Next i
    'Last macroblock in scan line
    'First field upper half ...
    clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 1, 1, 1, 1, 1, 1)
    '... Second field upper half ...
    clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 1, 1, 1, 1, 1, 0)
    '... First field lower half ...
    clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 1, 1, 1, 0)
    '... Second field lower half ...
    clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 1, 1, 1, 0)
    '... and similarly for the other two predictors:
    clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 1, 1, 1, 1, 1, 0)
    clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 1, 1, 1, 1, 1, 0)
    clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 1, 1, 1, 0)
    clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 1, 1, 1, 0)
    'Writing 4 storage units to memory:
    'First writes in each field ...
    clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 1, 1, 1, 1)
    clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 0, 1, 1, 0)
    'Second writes in each field without page crossing (no precharge)
    clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 0, 0, 1, 0, 0)
    clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 0, 0, 0, 0, 0)
Case 128
    'A storage unit is 2 adjascent 8 by 8 field blocks.  Four such st. units
    'are accessed to form one predictor.  Each predictor, given the mem.
```

```
'Function macro: pred_wc_clks(    b_width, latency, stu_clks, mbs_per_ln  )
'
'This function computes the number of clock cycles required per H reset
'to read in 4 (worst case) predictors from memory, and write each macroblock
'computed in that interval.  By definition, each pair of predictors is accessed
'from opposite banks of memory, so overhead can be saved.  Variables b_width
'and latency denote the bus width and CAS latency of the target SDRAM.
'Variable stu_clks represents the number of memory cycles required to access
'one storage unit in memory.  Variable mbs_per_ln represents the number of
'macroblocks processed per H reset

Function pred_wc_clks(b_width, latency, stu_clks, mbs_per_ln)

    clk_cnt = 0

    Select Case b_width
    Case 64
        'A storage unit is one 8 by 8 field block.  Six such st. units are
        'accessed to form one predictor.  Each predictor, given the memory
        'organization, can have at most one page crossing (in the upper 3
        'or lower 3 blocks but not both at the same time)

        'First macroblock in scan line ...
        'First field upper half ...
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 1, 1, 0, 1, 1, 1)
        '... Second field upper half ...
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 1, 1, 1, 1, 1, 0)
        '... First field lower half ...
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 1, 1, 1, 0)
        '... Second field lower half ...
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 1, 1, 1, 0)
        '... and similarly for the other two predictors:
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 1, 1, 1, 1, 1, 0)
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 1, 1, 1, 1, 1, 0)
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 1, 1, 1, 0)
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 1, 1, 1, 0)
        'Writing 4 storage units to memory:
        'Writes are not large enough to hide overhead
        'First writes in each field ...
        clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 1, 1, 1, 1)
        clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 0, 1, 1, 0)
        'Second writes in each field with page crossing (need precharge)
        clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 0, 1, 1, 0)
        clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 0, 1, 1, 0)

        'Intermediete macroblocks in scan line ....
        clk_cnt = clk_cnt + read_clks((osd_per_ln - osd_rd_cnt), latency, 0, 1, 0, 1, 1, 0)
    End If
    If osd_wr_cnt < osd_per_ln Then
        clk_cnt = clk_cnt + write_clks((osd_per_ln - osd_wr_cnt), latency, 0, 1, 0, 1, 1, 0)
    End If
    If vld_rd_cnt < vld_per_ln Then
        clk_cnt = clk_cnt + read_clks((vld_per_ln - vld_rd_cnt), latency, 0, 1, 0, 1, 1, 0)
    End If
    If vld_wr_cnt < vld_per_ln Then
        clk_cnt = clk_cnt + write_clks((vld_per_ln - vld_wr_cnt), latency, 0, 1, 0, 1, 1, 0)
    End If

    osd_vld_rd_wr = clk_cnt

End Function
```

```
While ((osd_rd_cnt + 8) <= osd_per_ln) And ((vld_rd_cnt + 8) <= vld_per_ln)
    'Worst case both reads are inserted between two writes.  But we assume
    'more intelligence on the part of the LMC.   Note that this worst case
    'situation cannot be represented in this model as the trailing write has
    'no idea about these read operations
    clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 1, 1, 1, 1)
    clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 1, 1, 1, 0)
    osd_rd_cnt = osd_rd_cnt + 8
    vld_rd_cnt = vld_rd_cnt + 8
Wend
While ((osd_wr_cnt + 8) <= osd_per_ln) And ((vld_wr_cnt + 8) <= vld_per_ln)
    'See comments above
    clk_cnt = clk_cnt + write_clks(8, latency, 0, 1, 1, 1, 1, 1)
    clk_cnt = clk_cnt + write_clks(8, latency, 0, 1, 1, 1, 1, 0)
    osd_wr_cnt = osd_wr_cnt + 8
    vld_wr_cnt = vld_wr_cnt + 8
Wend
While ((osd_wr_cnt + 8) <= osd_per_ln) And ((vld_rd_cnt + 8) <= vld_per_ln)
    clk_cnt = clk_cnt + write_clks(8, latency, 0, 1, 1, 1, 1, 1)
    clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 1, 1, 1, 1)
    osd_wr_cnt = osd_wr_cnt + 8
    vld_rd_cnt = vld_rd_cnt + 8
Wend
While ((osd_rd_cnt + 8) <= osd_per_ln) And ((vld_wr_cnt + 8) <= vld_per_ln)
    clk_cnt = clk_cnt + write_clks(8, latency, 0, 1, 1, 1, 1, 1)
    clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 1, 1, 1, 1)
    osd_rd_cnt = osd_rd_cnt + 8
    vld_wr_cnt = vld_wr_cnt + 8
Wend
While (osd_rd_cnt + 8) <= osd_per_ln
    clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 0, 1, 1, 0)
    osd_rd_cnt = osd_rd_cnt + 8
Wend
While (osd_wr_cnt + 8) <= osd_per_ln
    clk_cnt = clk_cnt + write_clks(8, latency, 0, 1, 0, 1, 1, 0)
    osd_wr_cnt = osd_wr_cnt + 8
Wend
While (vld_rd_cnt + 8) <= vld_per_ln
    clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 0, 1, 1, 0)
    vld_rd_cnt = vld_rd_cnt + 8
Wend
While (vld_wr_cnt + 8) <= vld_per_ln
    clk_cnt = clk_cnt + write_clks(8, latency, 0, 1, 0, 1, 1, 0)
    vld_wr_cnt = vld_wr_cnt + 8
Wend
If osd_rd_cnt < osd_per_ln Then
```

```
'Function macro: osd_vld_rd_wr(  osd_per_ln, vld_per_ln, latency    )
'
'This function computes the number of cycles required to perform paired
'read-write operations of OSD and VLD data.  Each line, a portion of data from
'a bitstream input FIFO is transferred to SDRAM.  At the same time, an OSD
'input FIFO is also serviced if data is available.  The two transfers occur in
'opposite memory banks.  So overhead can be hidden, provided each session is 8
'bus cycles long.  Variables osd_per_ln and vld_per_ln denote the number of OSD
'and VLD read and write cycles that are needed per scan line.  Variable latency
'denotes SDRAM CAS latency.

Function osd_vld_rd_wr(osd_per_ln, vld_per_ln, latency)

    clk_cnt = 0
    osd_rd_cnt = 0
    osd_wr_cnt = 0
    vld_rd_cnt = 0
    vld_wr_cnt = 0

    'No matter what the latency, PRE-ACTIVE (3 clks) and ACTIVE-READ/WRITE
    '(3 clks) will mandate 8 access cycles for zero overhead bank mux'ed
    'operations.  The paired bank mux'ed operations should escape from first
    'access and last access overhead if the LMC algorithm is synchronous enough
    'First access to account for page hit overhead
    If (osd_per_ln > 8) Then
        clk_cnt = clk_cnt + read_clks(8, latency, 1, 0, 1, 1, 1, 1)
        clk_cnt = clk_cnt + write_clks(8, latency, 1, 0, 1, 1, 1, 1)
        osd_rd_cnt = osd_rd_cnt + 8
        osd_wr_cnt = osd_wr_cnt + 8
    Else
        clk_cnt = clk_cnt + read_clks(osd_per_ln, latency, 1, 0, 0, 1, 1, 1)
        clk_cnt = clk_cnt + write_clks(osd_per_ln, latency, 1, 0, 0, 1, 1, 1)
        osd_rd_cnt = osd_per_ln
        osd_wr_cnt = osd_per_ln
    End If
    If (vld_per_ln > 8) Then
        clk_cnt = clk_cnt + read_clks(8, latency, 1, 0, 1, 1, 1, 0)
        clk_cnt = clk_cnt + write_clks(8, latency, 1, 0, 1, 1, 1, 0)
        vld_rd_cnt = vld_rd_cnt + 8
        vld_wr_cnt = vld_wr_cnt + 8
    Else
        clk_cnt = clk_cnt + read_clks(vld_per_ln, latency, 1, 0, 0, 1, 1, 0)
        clk_cnt = clk_cnt + write_clks(vld_per_ln, latency, 1, 0, 0, 1, 1, 0)
        vld_rd_cnt = vld_per_ln
        vld_wr_cnt = vld_per_ln
    End If
```

```
'Function macro: vld_clks(  b_width, bitrate, frame_rate, frame_lines   )
'
'This function computes the number of memory cycles required per
'scan line to satisfy reading or writing of compressed bitstream
'from or to the bit buffer

Function vld_clks(b_width, bitrate, frame_rate, frame_lines)

    If (Int(bitrate / (b_width * frame_rate * frame_lines))) < (bitrate / (b_width * frame_rate * frame_lines)) Th
en
        vld_clks = Int(bitrate / (b_width * frame_rate * frame_lines)) + 1
    Else
        vld_clks = bitrate / (b_width * frame_rate * frame_lines)
    End If

End Function
```

```
            clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 1, 1, 1, 1)
            clk_cnt = clk_cnt + write_clks((stu_per_ln - write_cnt), latency, 0, 1, 1, 0, 1, 1)
            clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 1, 1, 1, 1)
            clk_cnt = clk_cnt + write_clks((stu_per_ln - write_cnt), latency, 0, 1, 1, 0, 1, 1)
            '... one chroma access
            clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 1, 1, 1, 1)
            clk_cnt = clk_cnt + write_clks((stu_per_ln - write_cnt), latency, 0, 1, 1, 0, 1, 1)
            rd_pel_cnt = rd_pel_cnt + 256
            wr_pel_cnt = wr_pel_cnt + (stu_per_ln - write_cnt) * 32
        Wend
        If rd_pel_cnt < act_ln_pels Then
            clk_cnt = clk_cnt + read_clks(((act_ln_pels - rd_pel_cnt) / 16), latency, 0, 1, 0, 1, 1, 1)
            clk_cnt = clk_cnt + read_clks(((act_ln_pels - rd_pel_cnt) / 32), latency, 0, 1, 0, 1, 1, 1)
        End If
        While wr_pel_cnt < (stu_per_ln * 128)
            'Two luma field block lines and ...
            clk_cnt = clk_cnt + write_clks(((stu_per_ln - write_cnt) * 2), latency, 0, 1, 0, 1, 1, 1)
            '...one chroma field block line
            clk_cnt = clk_cnt + write_clks((stu_per_ln - write_cnt), latency, 0, 1, 0, 1, 1, 1)
            wr_pel_cnt = wr_pel_cnt + (stu_per_ln - write_cnt) * 32
        Wend
    End Select
    raster_rd_wr = clk_cnt
End Function
```

```
            clk_cnt = clk_cnt + write_clks((stu_per_ln - write_cnt), latency, 0, 1, 1, 0, 1, 1)
            clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 1, 1, 1, 1)
            clk_cnt = clk_cnt + write_clks((stu_per_ln - write_cnt), latency, 0, 1, 1, 0, 1, 1)
            '... one chroma access
            clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 1, 1, 1, 1)
            clk_cnt = clk_cnt + write_clks((stu_per_ln - write_cnt), latency, 0, 1, 1, 0, 1, 1)
            rd_pel_cnt = rd_pel_cnt + 128
            wr_pel_cnt = wr_pel_cnt + (stu_per_ln - write_cnt) * 16
        Wend
        If rd_pel_cnt < act_ln_pels Then
            clk_cnt = clk_cnt + read_clks(((act_ln_pels - rd_pel_cnt) / 8), latency, 0, 1, 0, 1, 1, 1)
            clk_cnt = clk_cnt + read_clks(((act_ln_pels - rd_pel_cnt) / 16), latency, 0, 1, 0, 1, 1, 1)
        End If
        While wr_pel_cnt < (stu_per_ln * 64)
            'Two luma field block lines and ...
            clk_cnt = clk_cnt + write_clks(((stu_per_ln - write_cnt) * 2), latency, 0, 1, 0, 1, 1, 1)
            '...one chroma field block line
            clk_cnt = clk_cnt + write_clks((stu_per_ln - write_cnt), latency, 0, 1, 0, 1, 1, 1)
            wr_pel_cnt = wr_pel_cnt + (stu_per_ln - write_cnt) * 16
        Wend

    Case 128
        'Each storage unit is two horizontally adjascent field blocks
        'No matter what the latency, PRE-ACTIVE (3 clks) and ACTIVE-READ/WRITE
        '(3 clks) will mandate 8 access cycles for zero overhead bank mux'ed
        'operations.  The paired bank mux'ed operations should escape from first
        'access and last access overhead if the LMC algorithm is synchronous enough
        'Active line pels is always a multiple of 16
        While ((write_cnt + 8) <= stu_per_ln) And ((read_cnt + 8) <= (Int(act_ln_pels / 128)))
            'Assume worst case, that every pair induces a read<->write transition
            For luma_blk = 1 To 8 Step 1
                clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 1, 1, 1, 1)
                clk_cnt = clk_cnt + write_clks(8, latency, 0, 1, 1, 1, 1, 1)
            Next luma_blk
            For chroma_blk = 1 To 4 Step 1
                clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 1, 1, 1, 1)
                clk_cnt = clk_cnt + write_clks(8, latency, 0, 1, 1, 1, 1, 1)
            Next chroma_blk
            write_cnt = write_cnt + 8
            read_cnt = read_cnt + 8
            rd_pel_cnt = rd_pel_cnt + 1024
            wr_pel_cnt = wr_pel_cnt + 1024
        Wend
        While ((rd_pel_cnt + 256) <= act_ln_pels) And ((wr_pel_cnt + ((stu_per_ln - write_cnt) * 32)) <= (stu_per_
ln * 128))
            'Two luma accesses and ...
```

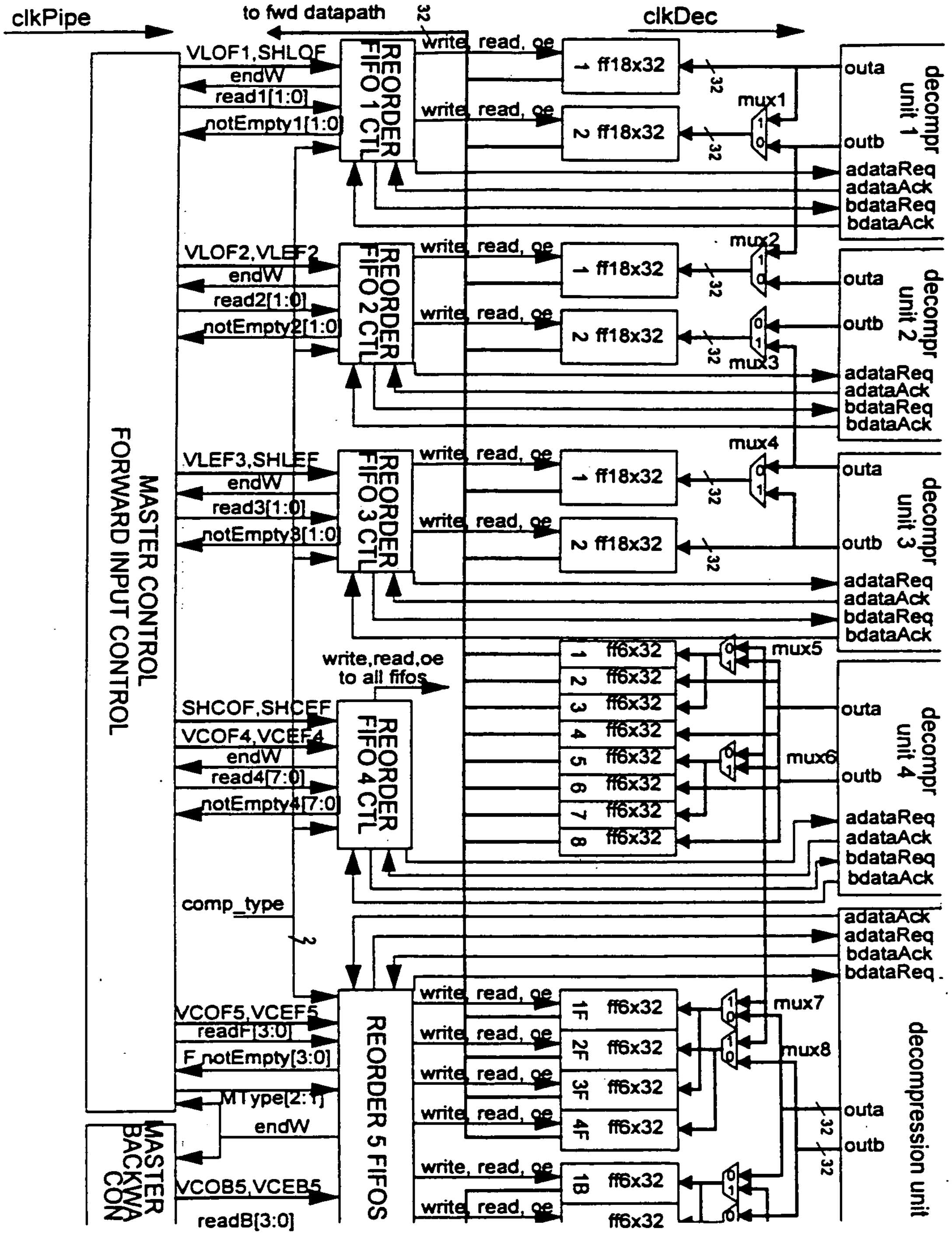
```
'Function macro: raster_rd_wr( act_ln_pels, bus_width, stu_per_ln, latency )
'
'This function computes the number of cycles required to perform paired
'read-write operations of 4:2:0 format display data.  Each line, a raster
'line of display luma and half a raster line of display chroma are read
'out of one bank, while the same number of luma and chroma pels are written
'into the opposite bank from the display section decode of stu_per_ln
'storage units.  The variable act_ln_pels represents the number of active luma
'pels in each line.  Variables bus_width and latency denote SDRAM bus width and
'CAS latency.

Function raster_rd_wr(act_ln_pels, bus_width, stu_per_ln, latency)

    clk_cnt = 0
    write_cnt = 0
    read_cnt = 0
    rd_pel_cnt = 0
    wr_pel_cnt = 0
    Select Case bus_width
    Case 64
        'Each storage unit is a field block - an 8 by 8 luma block with 2 4 by 4
        'chroma blocks
        'No matter what the latency, PRE-ACTIVE (3 clks) and ACTIVE-READ/WRITE
        '(3 clks) will mandate 8 access cycles for zero overhead bank mux'ed
        'operations.  The paired bank mux'ed operations should escape from first
        'access and last access overhead if the LMC algorithm is synchronous enough
        'Active line pels is always a multiple of 16
        While ((write_cnt + 8) <= stu_per_ln) And ((read_cnt + 8) <= (Int(act_ln_pels / 64)))
            'Assume worst case, that every pair induces a read<->write transition
            For luma_blk = 1 To 8 Step 1
                clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 1, 1, 1, 1)
                clk_cnt = clk_cnt + write_clks(8, latency, 0, 1, 1, 1, 1, 1)
            Next luma_blk
            For chroma_blk = 1 To 4 Step 1
                clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 1, 1, 1, 1)
                clk_cnt = clk_cnt + write_clks(8, latency, 0, 1, 1, 1, 1, 1)
            Next chroma_blk
            write_cnt = write_cnt + 8
            read_cnt = read_cnt + 8
            rd_pel_cnt = rd_pel_cnt + 512
            wr_pel_cnt = wr_pel_cnt + 512
        Wend
        While ((rd_pel_cnt + 128) <= act_ln_pels) And ((wr_pel_cnt + ((stu_per_ln - write_cnt) * 16)) <= (stu_per_
ln * 64))
            'Two luma accesses and ...
            clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 1, 1, 1, 1)
        Case 1
            Select Case res
            Case 0
                    frame_clks = pel_area / 32
            Case 1
                    frame_clks = pel_area / 64
            Case 2
                    frame_clks = pel_area / 96
            End Select
        End Select
    End Select
    If (Int(frame_clks / frame_lines)) < (frame_clks / frame_lines) Then
        osd_clks = Int(frame_clks / frame_lines) + 1
    Else
        osd_clks = frame_clks / frame_lines
    End If
End Function
```

```
'Function macro: osd_clks(  b_width, h_size, v_size, mode, res, frame_lines )
'
'This function computes the number of memory cycles required per
'scan line to satisfy reading or writing of OSD information from
'or to the SDRAM.  Variable b_width denotes the SDRAM bus bandwidth
'Variables h_size and v_size represent the worst case horizontal and
'vertical sizes of OSD.  Variable mode refers to 2 bit/pixel (=0) or
'4 bit/pixel (1) OSD.  Variable res refers to OSD resolution: 0=>full
';1=>half and 2=>one-third.  Variable frame_lines refers to the number
'of lines between two odd synchs

Function osd_clks(b_width, h_size, v_size, mode, res, frame_lines)
    pel_area = h_size * v_size
    Select Case b_width
    Case 64
        Select Case mode
        Case 0
            Select Case res
            Case 0
                    frame_clks = pel_area / 32
            Case 1
                    frame_clks = pel_area / 64
            Case 2
                    frame_clks = pel_area / 96
            End Select
        Case 1
            Select Case res
            Case 0
                    frame_clks = pel_area / 16
            Case 1
                    frame_clks = pel_area / 32
            Case 2
                    frame_clks = pel_area / 48
            End Select
        End Select
    Case 128
        Select Case mode
        Case 0
            Select Case res
            Case 0
                    frame_clks = pel_area / 64
            Case 1
                    frame_clks = pel_area / 128
            Case 2
                    frame_clks = pel_area / 192
            End Select
```

```
                clk_cnt = clk_cnt + read_clks(stu_clks, latency, 0, 1, 0, 1, 1, 0)
                num_pg_hits = num_pg_hits - 1
                blk_cnt = blk_cnt + 1
            Else
                'Very first access in a cycle is new page
                clk_cnt = clk_cnt + read_clks(stu_clks, latency, 0, 1, 0, 1, 1, 0)
                blk_cnt = blk_cnt + 1
            End If
            If blk_cnt < stu_per_ln Then
                clk_cnt = clk_cnt + read_clks((stu_clks * (stu_per_ln - blk_cnt)), latency, 0, 0, 1, 1, 0, 0)
                blk_cnt = stu_per_ln
            End If
        End If
    Next cyc_count
    blk2ras_dec = clk_cnt
End Function
```

```
'Function macro: blk2ras_dec(   stu_clks, latency, stu_per_pg, stu_per_ln, repeat_cnt  )
'
'This function computes the number of cycles required to decode stu_per_ln
'number of field blocks.  These field blocks form the next 8 field lines
'that will be displayed.  The variable stu_per_pg helps calculates the worst
'case number of page hits incurred in reading stu_per_ln field blocks.  The
'variable repeat_cnt is the number of times the basic memory access pattern
'is repeated during a line.  The variable stu_clks is the number of mem. cycles
'required to access one storage unit.  The variable latency is the SDRAM CAS
'latency

Function blk2ras_dec(stu_clks, latency, stu_per_pg, stu_per_ln, repeat_cnt)
    If (Int((stu_per_ln / stu_per_pg))) < (stu_per_ln / stu_per_pg) Then
        num_pg_hits = Int((stu_per_ln / stu_per_pg)) + 1
    Else
        num_pg_hits = stu_per_ln / stu_per_pg
    End If
    If (Int((stu_per_ln / repeat_cnt))) < (stu_per_ln / repeat_cnt) Then
        blks_per_cycle = Int((stu_per_ln / repeat_cnt)) + 1
    Else
        blks_per_cycle = stu_per_ln / repeat_cnt
    End If
    clk_cnt = 0
    blk_cnt = 0
    For cyc_count = 1 To repeat_cnt Step 1
        If (blk_cnt + blks_per_cycle) <= stu_per_ln Then
            If num_pg_hits > 0 Then
                'Very first access in a cycle is new page
                clk_cnt = clk_cnt + read_clks(stu_clks, latency, 0, 1, 0, 0, 1, 0)
                'Assuming blks_per_cycle does not span more than 1 page!
                clk_cnt = clk_cnt + read_clks(stu_clks, latency, 0, 1, 0, 1, 1, 0)
                num_pg_hits = num_pg_hits - 1
                If blks_per_cycle > 2 Then
                    clk_cnt = clk_cnt + read_clks((stu_clks * (blks_per_cycle - 2)), latency, 0, 0, 1, 1, 0, 0)
                End If
            Else
                'Very first access in a cycle is new page
                clk_cnt = clk_cnt + read_clks(stu_clks, latency, 0, 1, 0, 1, 1, 0)
                clk_cnt = clk_cnt + read_clks((stu_clks * (blks_per_cycle - 1)), latency, 0, 0, 1, 1, 0, 0)
            End If
            blk_cnt = blk_cnt + blks_per_cycle
        Else
            If num_pg_hits > 0 Then
                'Very first access in a cycle is new page
                clk_cnt = clk_cnt + read_clks(stu_clks, latency, 0, 1, 0, 0, 1, 0)
                blk_cnt = blk_cnt + 1
```

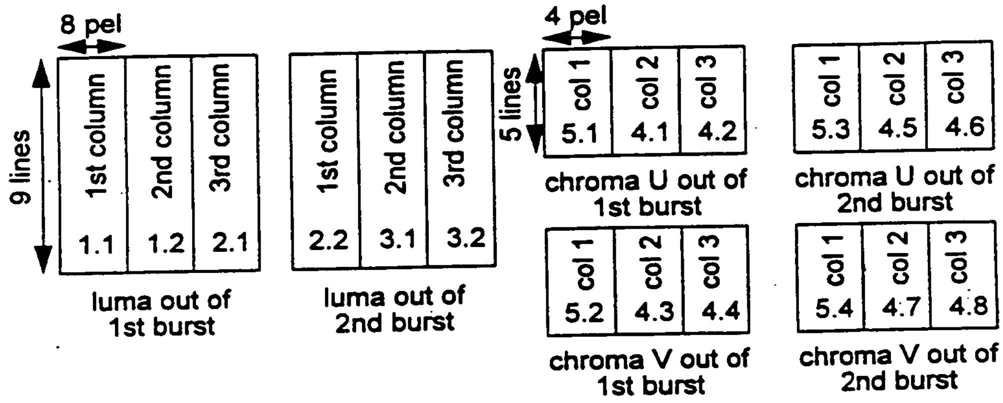
Block Diagram


clkPipe
to fwd datapath
32
clkDec
MASTER CONTROL
FORWARD INPUT CONTROL
VLOF1,SHLOF
endW
read1[1:0]
notEmpty1[1:0]
REORDER FIFO 1 CTL
write
read, oe
1 ff18x32
2 ff18x32
32
mux1
outa
outb
decompr unit 1
adataReq
adataAck
bdataReq
bdataAck
VLOF2,VLEF2
endW
read2[1:0]
notEmpty2[1:0]
REORDER FIFO 2 CTL
mux2
mux3
decompr unit 2
VLEF3,SHLEF
endW
read3[1:0]
notEmpty3[1:0]
REORDER FIFO 3 CTL
mux4
decompr unit 3
write,read,oe
to all fifos
SHCOF,SHCEF
VCOF4,VCEF4
endW
read4[7:0]
notEmpty4[7:0]
REORDER FIFO 4 CTL
ff6x32
mux5
mux6
decompr unit 4
comp_type
VCOF5,VCEF5
readF[3:0]
F notEmpty[3:0]
MType[2:1]
endW
VCOB5,VCEB5
readB[3:0]
REORDER 5 FIFOS
1F
2F
3F
4F
1B
mux7
mux8
decompression unit
MASTER BACKWA CON

6.3.b. Reordering fifos fonction

The reordering fifos allow pixel data to be collected after the decompression and to be read by the master controller in a suitable order for the datapath. There storage capacity is the total data needed to generate one macroblock prediction.

Each fifo is associated with a 8 pixel column (luma) or 4 pixel column (chroma) of data to be used for generating prediction. The next figure represent the fifo location for each column, depending on the memory burst the column comes out.



In no compression mode or 2M/3, columns of data directly comes from column of the burst. In H/2-M/2 mode, column are generated through the mux placed before certain fifos. In no compression or 2M/3, no horizontal selection is done : all horizontal columns are put in reordering fifos. In H/2-M/2 modes, as 4 columns are generated due to the H/2 compression mode, one is deleted while writing, using the horizontal vector. Moreover, as the decompression units used are not corresponding to the reordering fifos as in other modes, output of decompression units are redirected to appropriate fifos (through the muxes).

The other role of the reordering fifo controllers is also to delete, in 2M/3 and H/2-M/2, the vertical data that are not used to generate the predictors. They use the vertical component of the vector to throw away 7 lines over 16 for luma and 3 over 8 for chroma. A more detailed mode of operation will be presented later.

6.3.c. Reordering fifo controllers handshakes

- Reordering fifos 1 to 3 or 6 to 8 controller

Those decompression fifos controller generate one write request for each of their fifo (i.e. 2 request named adataReq and bdataReq). These request are equivalent to "fifo not full". These request are sent to the same decompression unit and correspond to request of the 2 interleaved different blocks. The first block to appear after decompression is always named "a" block and is sent to the "fifo 1" of the set of fifos.

The decompression block answer to the request with an acknowledge signal (see timing diagram). The clock on the input side of the fifos is clkDec (81 MHz). There can never be two consecutive writes to the same fifo.

- Reordering fifos 4 or 9 controller

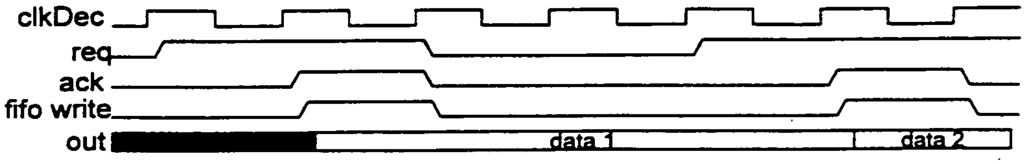
Those decompression fifo controller generate also 2 requests (a and b) but fill their 8 fifos in a specific order.

- Reordering fifos 5 controller

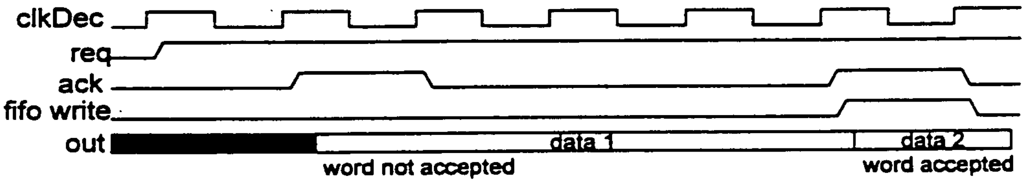
The number of fifos is 8 in this case, 4 for the forward path, 4 for the backward one. The "a" block out of the decompression unit can go to 4 different fifos (1F, 3F, 1B or 3B), the selection is done by the fifos controller and will be explained later.

On the output side of the fifos, all forward fifos are connected together, as well as all backward fifos. Read selection is made by the master control. Each time a fifo is read, its output is enabled.

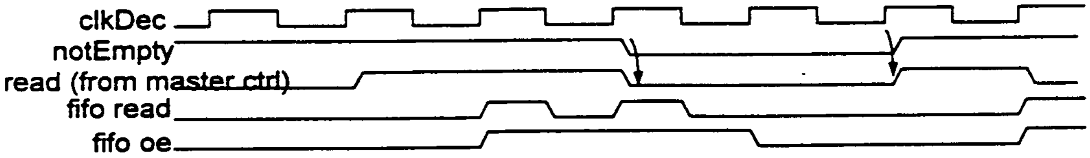
Fifos input timing



The write is not only controlled by the input ack but also through a little state machine that takes care on the vertical vector : some words are output by the decompression unit but not written to the fifos.



Fifos output timing (the clock here is clkPipe (54 MHz))



6.3.d. Reordering fifos control vector and macroblock type handshake

Each control unit get information from the associated (forward or backward) master controller. Each control unit counts the outputs of the associated decompression unit, write data to its fifos through a determined sequence (see next paragraph) and send a "endW" signal to the master controller when the last data of a macroblock is output from the decompression unit. This allow the master control to present new vector and if needed macroblock type information to the fifo controller.

The vector and macroblock type information needed are (between parenthesis is shown the compression modes where those information are effectively used) :

- for unit 1 :
  - VLOF1[3:1] vertical luma odd (top) field forward vector for unit 1 (2M/3,H/2-M/2)
  - SHLOF one bit shift horizontal luma odd (top) field forward (H/2-M/2)
- for unit 2 :
  - VLOF2[3:1] vertical luma odd (top) field forward vector for unit 2 (2M/3,H/2-M/2)
  - VLEF2[3:1] vertical luma even (bot) field forward vector for unit 2 (2M/3,H/2-M/2)
- for unit 3 :
  - VLEF3[3:1] vertical luma even (bot) field forward vector for unit 3 (2M/3,H/2-M/2)
  - SHLEF one bit shift horizontal luma even (bot) field forward (H/2-M/2)
- for unit 4 :
  - VCOF4[2:1] vertical chroma odd (top) field forw vector for unit 4 (2M/3,H/2-M/2)
  - VCEF4[2:1] vertical chroma even (bot) field forw vector for unit 4 (2M/3,H/2-M/2)
  - SHCOF one bit shift horizontal chroma odd (top) field forward (H/2-M/2)
  - SHCEF one bit shift horizontal chroma even (bot) field forward (H/2-M/2)
- for unit 5 :
  - MBType[2:1] macroblock type (all comp mode)
  - VCOF5[2:1] vertical chroma odd (top) field forw vector for unit 4 (2M/3,H/2-M/2)
  - VCEF5[2:1] vertical chroma even (bot) field forw vector for unit 4 (2M/3,H/2-M/2)
  - VCOB5[2:1] vertical chroma odd (top) field backw vect for unit 4 (2M/3,H/2-M/2)
  - VCEB5[2:1] vertical chroma even (bot) field backw vect for unit 4 (2M/3,H/2-M/2)

The master controller description will show the way to obtain those informations from the VLD Vbus data. From the fifo controller point of view, these are static information changing between macroblocks.

6.3.e. Reordering fifos write control

- No compression mode :

All muxes (1 to 8 for forward direction) are in default mode (i.e. input '0' selected). No data is discarded at this stage (i.e. all data input in decompression fifos are written in reordering fifos).

The order of write for a macroblock is :

- unit 1 : outa -> 18 writes in fifo 1.1 ; outb -> 18 writes in fifo 1.2
- unit 2 : outa -> 18 writes in fifo 2.1 ; outb -> 18 writes in fifo 2.2
- unit 3 : outa -> 18 writes in fifo 3.1 ; outb -> 18 writes in fifo 3.2
- unit 4 : outa -> 10 alternate writes to 4.1 and 4.3, starting with 4.1, then 10 alternate writes to 4.5 and 4.7, starting with 4.5 ;
  outb -> 10 alternate writes to 4.2 and 4.4, starting with 4.2, then 10 alternate writes to 4.6 and 4.8, starting with 4.6
- unit 5 : the output depends on the MB type
  outa ->
  - if forward or bidirectional : 5 writes in 5.1F, then
  - if backward or bidirectional : 5 writes in 5.1B then
  - if forward or bidirectional : 5 writes in 5.3F, then
  - if backward or bidirectional : 5 writes in 5.3B

  outb ->
  - if forward or bidirectional : 5 writes in 5.2F, then
  - if backward or bidirectional : 5 writes in 5.2B then

- if forward or bidirectional : 5 writes in 5.4F, then
- if backward or bidirectional : 5 writes in 5.4B

- 2M/3 compression :

All muxes (1 to 8 for forward direction) are in default mode (i.e. input '0' selected). Only vertical data are discarded at this stage : luma column are reduced from 16 lines to 9, chroma from 8 to 5.

The order of write for a macroblock is :

- unit 1 : outa -> 2*VLOF1 non written data, then 18 writes in fifo 1.1,
  then 2*not(VLOF1) non written data (or complete to 32, as total number of output data is 32)
  outb -> 2*VLOF1 non written data, then 18 writes in fifo 1.2,
  then 2*not(VLOF1) non written data (or complete to 32)

- unit 2 : outa -> 2*VLOF2 non written data, then 18 writes in fifo 2.1,
  then 2*not(VLOF2) non written data (or complete to 32)
  outb -> 2*VLEF2 non written data, then 18 writes in fifo 2.2,
  then 2*not(VLEF2) non written data (or complete to 32)

- unit 3 : 2*VLEF3 non written data, then 18 writes in fifo 3.1,
  then 2*not(VLEF3) non written data (or complete to 32)
  outb -> 2*VLEF3 non written data, then 18 writes in fifo 3.2,
  then 2*not(VLEF3) non written data (or complete to 32)

- unit 4 : outa -> (total 32 data output)
  - VCOF4 non written data, then (4 - VCOF4) writes to 4.1 (or complete to 4) (U3)
  - VCOF4 non written data, then (4 - VCOF4) writes to 4.3 (or complete to 4) (V3)
  - (VCOF4 + 1) writes to 4.1, then complete to 4 with non written data (U4)
  - (VCOF4 + 1) writes to 4.3, then complete to 4 with non written data (V4)
  - VCEF4 non written data, then (4 - VCEF4) writes to 4.5 (or complete to 4) (U9)
  - VCEF4 non written data, then (4 - VCEF4) writes to 4.7 (or complete to 4) (V9)
  - (VCEF4 + 1) writes to 4.5, then complete to 4 with non written data (U10)
  - (VCEF4 + 1) writes to 4.7, then complete to 4 with non written data (V10)

  outb -> (total 32 data output)
  - VCOF4 non written data, then (4 - VCOF4) writes to 4.2 (or complete to 4) (U5)
  - VCOF4 non written data, then (4 - VCOF4) writes to 4.4 (or complete to 4) (V5)
  - (VCOF4 + 1) writes to 4.2, then complete to 4 with non written data (U6)
  - (VCOF4 + 1) writes to 4.4, then complete to 4 with non written data (V6)
  - VCEF4 non written data, then (4 - VCEF4) writes to 4.6 (complete to 4) (U11)
  - VCEF4 non written data, then (4 - VCEF4) writes to 4.8 (complete to 4) (V11)
  - (VCEF4 + 1) writes to 4.6, then complete to 4 with non written data (U12)
  - (VCEF4 + 1) writes to 4.8, then complete to 4 with non written data (V12)

- unit 5 : the output depends on the MB type outa -> (total 16 data out if monodirectional, 32 if bidirectional)

- if forward or bidirectional : VCOF5 non written data, then 5 writes to 5.1F, then complete to 8 with non written data (U1, U2 forward), then
- if backward or bidirectional : VCOB5 non written data, then 5 writes to 5.1B, then complete to 8 with non written data (U1, U2 backward), then
- if forward or bidirectional : VCEF5 non written data, then 5 writes to 5.3F, then complete to 8 with non written data (U7, U8 forward), then
- if backward or bidirectional : VCEB5 non written data, then 5 writes to 5.3B, then complete to 8 with non written data (U7, U8 backward)

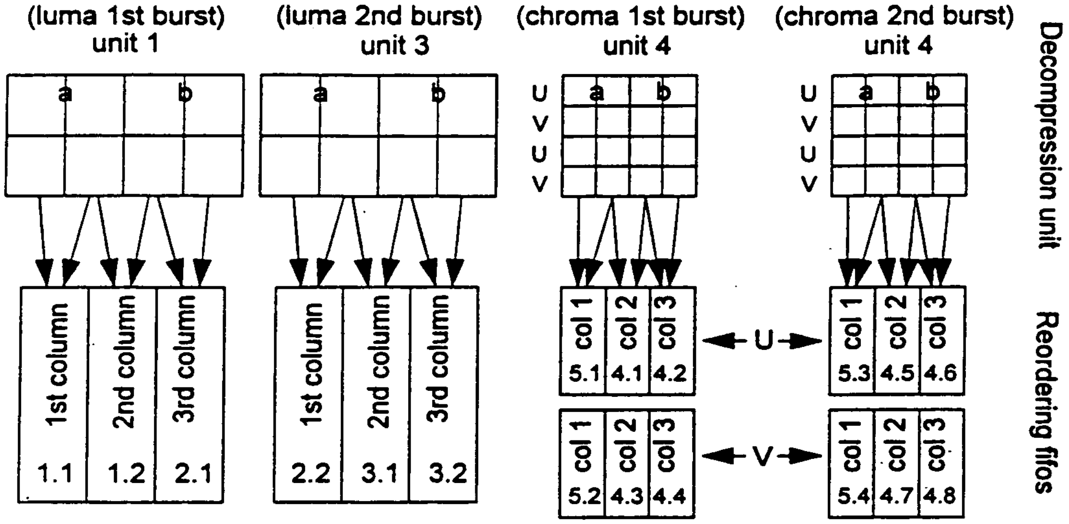
outb -> (total 16 data out if monodirectional, 32 if bidirectional)

- if forward or bidirectional : VCOF5 non written data, then 5 writes to 5.2F, then complete to 8 with non written data (V1, V2 forward), then
- if backward or bidirectional : VCOB5 non written data, then 5 writes to 5.2B, then complete to 8 with non written data (V1, V2 backward), then
- if forward or bidirectional : VCEF5 non written data, then 5 writes to 5.4F, then complete to 8 with non written data (V7, V8 forward), then
- if backward or bidirectional : VCEB5 non written data, then 5 writes to 5.4B, then complete to 8 with non written data (V7, V8 backward)

- H/2-M/2 compression :

Muxes (2,3,7,8) select input '1' (i.e. selection of those muxes is "H/2-M/2 mode").
Mux 1 selects input '1' if SHLOF is low (selection of mux 1 is H/2-M/2 and notSHLOF)
Mux 4 selects input '1' if SHLEF is high (selection of mux 4 is H/2-M/2 and SHLEF)
Mux 5 selects input '1' if SHCOF is high (selection of mux 5 is H/2-M/2 and SHCOF)
Mux 6 selects input '1' if SHCEF is high (selection of mux 1 is H/2-M/2 and SHCEF)

Vertical and horizontal data are discarded at this stage : luma column are reduced from 16 lines to 9, chroma from 8 to 5, and number of block column are reduced in this mode from 8 per direction to 6 (in this mode, each decompression block in fact corresponds to a 16 pels x 8 lines rectangle, so as 2 horizontally adjacent pixel blocks).



Selection of the data flow (arrow) is done through the value of :

SHLOF SHLEF SHCOF SHCEF

The order of write for a macroblock is :

- unit 1 : (64 data per output))
  - if SHLOF is low :
    outa -> 4*VLOF1 non written data, then alternate 2 writes in fifo 1.1 with 2 writes in fifo 1.2, starting with 1.1, until 18 writes are done in each 1.1 and 1.2, then complete to 64 reads with non written data.
    outb -> 4*VLOF1 non written data, then alternate 2 writes in fifo 2.1 with 2 non written data, until 18 writes are done in fifo 2.1, then complete to 64 reads with non written data.
  - if SHLOF is high :
    outa -> 4*VLOF1 non written data, then alternate 2 non written data with 2 writes in fifo 1.1, until 18 writes are done in 1.1, then complete to 64 reads with non written data.
    outb -> 4*VLOF1 non written data, then alternate 2 writes in fifo 1.2 with 2 writes in fifo 2.1 until 18 writes are done in each 1.2 and 2.1, then complete to 64 reads with non written data.

- unit 2 : not used.

- unit 3 : (64 data per output)
  - if SHLEF is low :
    outa -> 4*VLEF3 non written data, then alternate 2 writes in fifo 2.2 with 2 writes in fifo 3.1, starting with 2.2, until 18 writes are done in each 2.2 and 3.1, then complete to 64 reads with

non written data.
outb -> 4*VLEF3 non written data, then alternate 2 writes in fifo 3.2 with 2 non written data, until 18 writes are done in fifo 3.2, then complete to 64 reads with non written data.

- if SHLOF is high :
  outa -> 4*VLEF3 non written data, then alternate 2 non written data with 2 writes in fifo 2.2, until 18 writes are done in 2.2, then complete to 64 reads with non written data.
  outb -> 4*VLEF3 non written data, then alternate 2 writes in fifo 3.1 with 2 writes in fifo 3.2 until 18 writes are done in each 3.1 and 3.2, then complete to 64 reads with non written data.

- unit 4 : (64 data per output)
  First 32 data of each output :
  - If SHCOF is low :
    outa ->
    2*VCOF4 non written data, then alternate writes in fifo 5.1 with writes in fifo 4.1 (starting with 5.1) until 8 data are output (U), then
    2*VCOF4 non written data, then complete to 8 by alternating writes in fifos 5.2 and 4.3 (V), then
    2*(VCOF4 + 1) alternate writes in 5.1 and 4.1, then complete to 8 with non written data (U, 5 data have been written to each 5.1 and 4.1), then
    2*(VCOF4 + 1) alternate writes in 5.2 and 4.3, then complete to 8 with non written data (V, 5 data have written to each 5.2 and 4.3)
    outb ->
    2*VCOF4 non written data, then complete to 8 by alternating one write in 4.2 with one non written data (U), then
    2*VCOF4 non written data, then complete to 8 by alternating one write in 4.4 with one non written data (V), then
    2*(VCOF4 + 1) alternate writes in 4.2 and non writes, then complete to 8 with non written data (U, 5 data have been written to 4.2), then
    2*(VCOF4 + 1) alternate writes in 4.4 and non writes, then complete to 8 with non written data (V, 5 data have been written to 4.4)

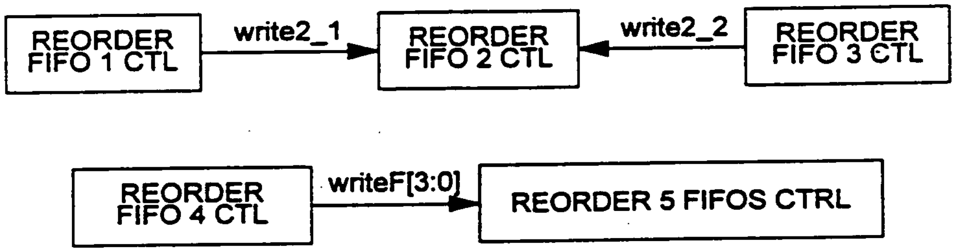
- If SHCOF is high :
  outa ->
  2*VCOF4 non written data, then alternate non writes with writes in fifo 5.1 until 8 data are output (U), then
  2*VCOF4 non written data, then complete to 8 by alternating non writes with writes in fifos 5.2 (V), then
  2*(VCOF4 + 1) alternate non writes and writes in 5.1, then complete to 8 with non written data (U, 5 data have been written to 5.1), then
  2*(VCOF4 + 1) alternate non writes and writes in 5.2, then complete to 8 with non written data (V, 5 data have written to each 5.2)
  outb ->
  2*VCOF4 non written data, then complete to 8 by alternating writes in 4.1 and writes in 4.2 (U), then
  2*VCOF4 non written data, then complete to 8 by alternating writes in 4.3 and writes in 4.4 (V), then
  2*(VCOF4 + 1) alternate writes in 4.1 and 4.2, then complete to 8 with non written data (U, 5 data have been written to each 4.1 and 4.2), then
  2*(VCOF4 + 1) alternate writes in 4.3 and 4.4, then complete to 8 with non written data (V, 5 data have been written to each 4.3 and 4.4)

- Last 32 data of each output :
  - If SHCEF is low :
    outa ->
    2*VCEF4 non written data, then alternate writes in fifo 5.3 with writes in fifo 4.5 (starting with 5.3) until 8 data are output (U), then
    2*VCEF4 non written data, then complete to 8 by alternating writes in fifos 5.4 and 4.7 (V), then
    2*(VCEF4 + 1) alternate writes in 5.3 and 4.5, then complete to 8 with non written data (U, 5 data have been written to each 5.3 and 4.5), then
    2*(VCEF4 + 1) alternate writes in 5.4 and 4.7, then complete to 8 with non written data (V, 5 data have written to each 5.4 and 4.7)
    outb ->
    2*VCEF4 non written data, then complete to 8 by alternating one write in 4.6 with one non written data (U), then
    2*VCEF4 non written data, then complete to 8 by alternating one write in 4.8 with one non written data (V), then
    2*(VCEF4 + 1) alternate writes in 4.6 and non writes, then complete to 8 with non written data (U, 5 data have been written to 4.6), then
    2*(VCEF4 + 1) alternate writes in 4.8 and non writes, then complete to 8 with non written data (V, 5 data have been written to 4.8)
  - If SHCEF is high :
    outa ->
    2*VCEF4 non written data, then alternate non writes with writes in fifo 5.3 until 8 data are output (U), then
    2*VCEF4 non written data, then complete to 8 by alternating non writes with writes in fifos 5.4 (V), then

2*(VCEF4 + 1) alternate non writes and writes in 5.3, then complete to 8 with non written data (U, 5 data have been written to 5.3), then
2*(VCEF4 + 1) alternate non writes and writes in 5.4, then complete to 8 with non written data (V, 5 data have written to each 5.4)
outb ->
2*VCOF4 non written data, then complete to 8 by alternating writes in 4.5 and writes in 4.6 (U), then
2*VCOF4 non written data, then complete to 8 by alternating writes in 4.7 and writes in 4.8 (V), then
2*(VCOF4 + 1) alternate writes in 4.5 and 4.6, then complete to 8 with non written data (U, 5 data have been written to each 4.5 and 4.6), then
2*(VCOF4 + 1) alternate writes in 4.7 and 4.8, then complete to 8 with non written data (V, 5 data have been written to each 4.7 and 4.8)

- unit 5 : not used

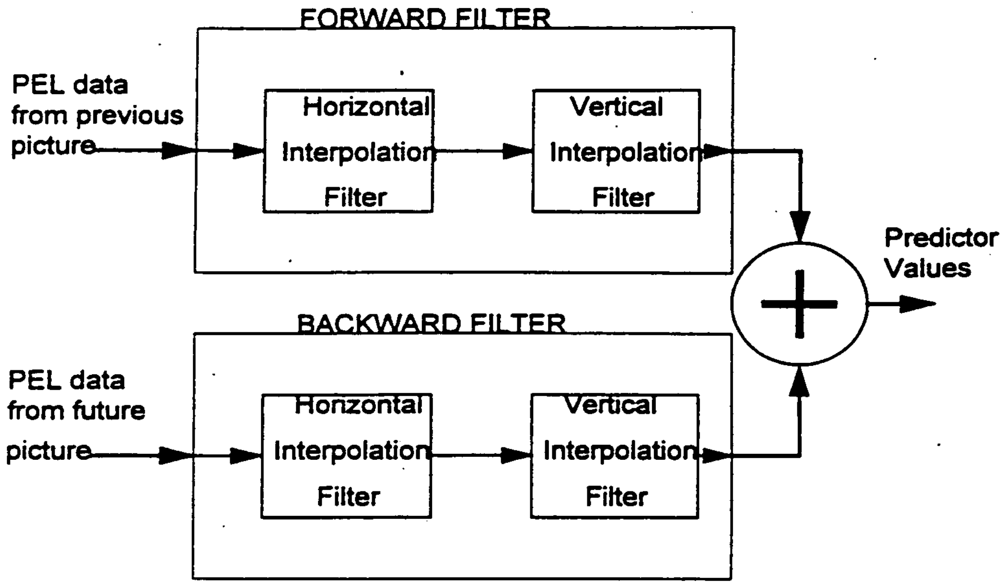
To allow the correct mechanism in H/2-M/2 mode, there is an additional communication between fifo controllers :



Reorder 1 and 3 controller send write envelopp signal to reorder 2 for its fifo, and reorder 4 do the same for the fifos F of reorder 5. Reorder 2 and 5 continue to control their own fifos.

7. *Datapath description*

The data path consists of two parallel paths that compute the forward and backward predictor values for each for each picture element in the 18 X 24 space for luma and the 9 X 24 space for chroma (Cb and Cr are handled separately). The two paths are then merged in a final combining summing cell. A simplified block diagram of the datapath is shown here:


FORWARD FILTER
PEL data from previous picture
Horizontal Interpolation Filter
Vertical Interpolation Filter
Predictor Values
BACKWARD FILTER
PEL data from future picture
Horizontal Interpolation Filter
Vertical Interpolation Filter

The PEL data from previous and future display pictures are read from memory, decompressed and fed to an input buffer. The control section recognises the fifo is full and starts retrieving data from the forward and backward fifos. PEL data is 8 bits wide for both luma and chroma. The data path is 32 bits wide and as so acts on 4 pel words at a time.

7.1. Horizontal interpolation filter

7.1.a. Description

The horizontal interpolation filter is used to compute predictor values based on input vectors in half pel multiples. The lower three bits of the horizontal motion vector are fed to the horizontal interpolation filter. These bits specify a vector of magnitude 0 to 3.5 half pels. Only the last three bits of the horizontal motion vector is dealt with in the horizontal filter. The more significant bits are used to generate addressing, thus the data reaching the horizontal filter is within one and one half (3.5) pels of the required predictor values.

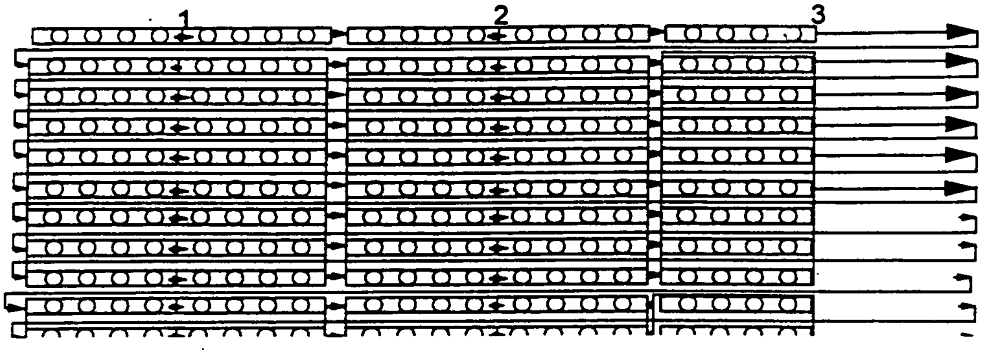
7.1.b. Input data format

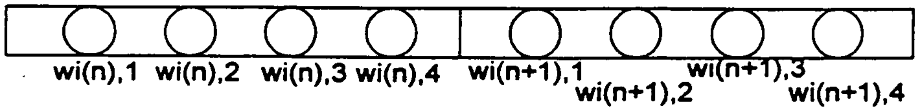
In order to understand the action of the horizontal filter it is necessary to understand how the data is presented to the filter. The diagram shows the order the data is clocked though the filter.

THOMSON CONSUMER ELECTRONICS CONFIDENTIAL AND PROPRIETARY

These drawings and specifications are the property of Thomson Consumer Electronics Inc. and shall not be reproduced or copied or used as the basis for manufacture or sale of apparatus or devices without permission.

Filter presentation order (Luma example)





wi(n) – word in n
wi(n+1) – word in n plus 1

Using this scheme the table in the next section describes the action of the filter for various values of the horizontal motion vector.

7.1.c. Horizontal filter hardware description.

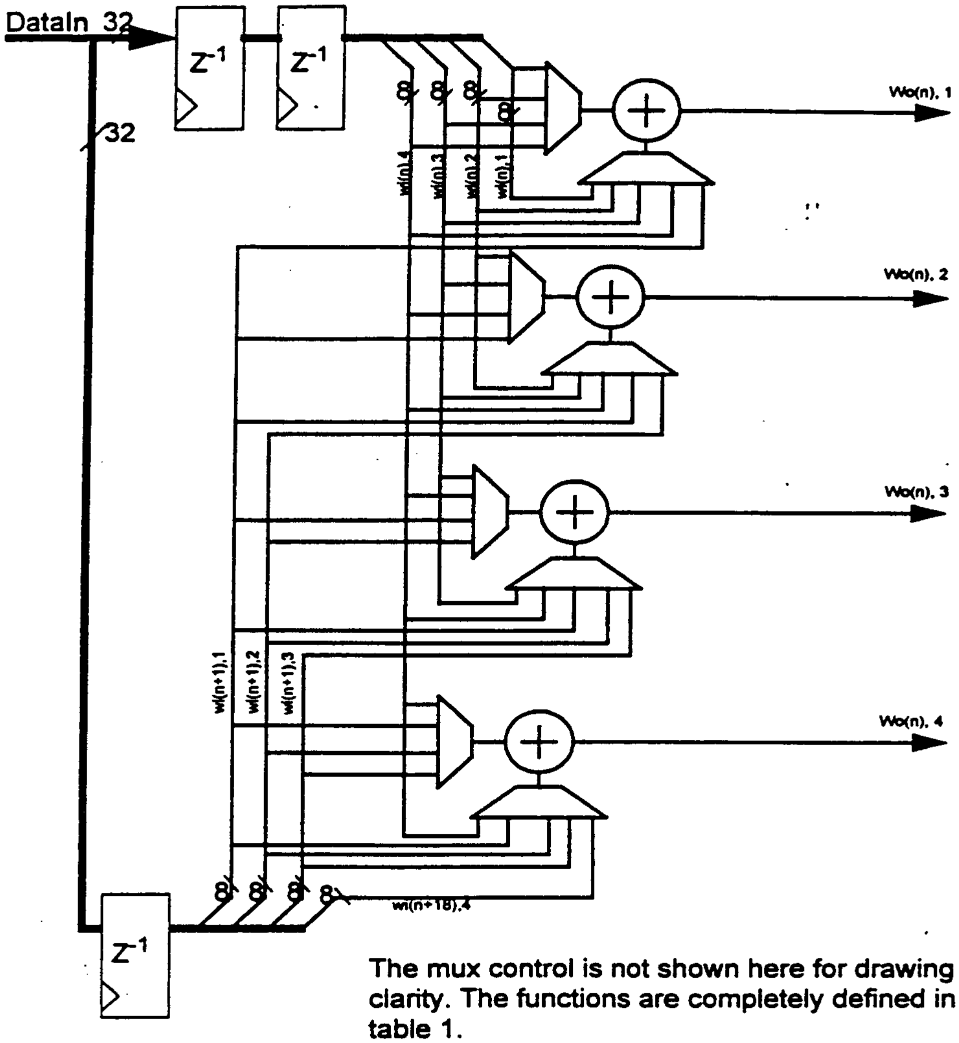
Table 4: Horizontal Motion vector actions

Motion vector	Vector value	Calculations	Description
0.0 pel	000	Wi n,1 + Wi n,1 = Wo n,1 Wi n,2 + Wi n,2 = Wo n,2 Wi n,3 + Wi n,3 = Wo n,3 Wi n,4 + Wi n,4 = Wo n,4	Copy from current position.
0.5 pel	001	Win,1 + Win,2 = Won,1 Win,2 + Win,3 = Won,2 Win,3 + Wn,4 = Won,3 Win,4 + Win+1,1 = Won,4	Interpolate with right adjacent pixel.
1.0pel	010	Win,2 + Win,2 = Won,1 Win,3 + Win,3 = Won,2 Win,4 + Win,4 = Won,3 Win+1,1 + Win+1,1 = Won,4	Copy from adjacent pixel.
1.5 pel	011	Win,2 + Wn,3 = Won,1 Win,3 + Win,4 = Won,2 Win,4 + Win+1,1 = Won,3 Win+1,1 + Win+1,2 = Won,4	Interpolate between two right adjacent pixels.
2.0 pel	100	Win,3 + Win3 = Won, 1 Win,4 + Win4 = Won,2 Win+1,1 + Win+1,1 = Won, 3 Win+1,2 + Win+1,2 = Won,4	Copy from 2 pixels to the right.

Table 4: Horizontal Motion vector actions

Motion vector	Vector value	Calculations	Description
2.5pel	101	Win,3 + Win4 = Won, 1 Win,4 + Win1,1 = Won,2 Win+1,1 + Win+1,2 = Won,3 Win+1,2 + Win+1,3 = Won,4	Interpolate between the 2nd and 3rd pixels to the right
3.0pel	110	Win,4 + Win4 = Won, 1 Win+1,1 + Win1,1 = Won,2 Win+1,2 + Win+1,2 = Won,3 Win+1,3 + Win+1,3 = Won,4	Copy from third pixel to the right.
3.5pel	111	Win,4 + Win+1,1 = Won,1 Win+1,1 + Win1,2 = Won,2 Win+1,2 + Win+1,3 = Won,3 Win+1,3 + Win+1,4 = Won,4	Interpolate between the 3rd and 4th pixel to the right.

From the table it can been seen that all operations result in an increase in bit with from 8 to 9. It can also be seen that 4 values are computed at a time. Note that 8 adjacent values are necessary to compute the 4 predictor values. The hardware to implement this is shown below:



The mux control is not shown here for drawing clarity. The functions are completely defined in table 1.

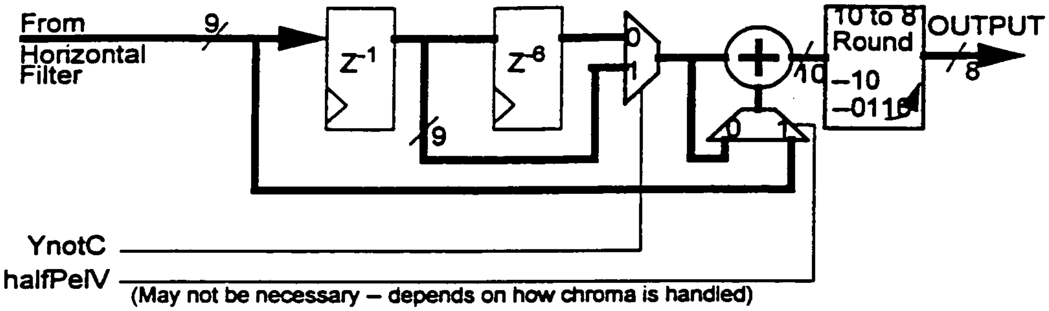
7.2. Vertical Filter

7.2.a. Vertical Filter Functional description

The vertical filter is responsible for calculating the vertical predictors from a motion vector of 0 or 0.5 pel. Under the data flow shown in figure 2 this is accomplished by using a delay of 5 element to store the vertical adjacent. For a motion vector of zero the predictor is the original value. For a motion vector of 0.5 the predictor value is calculated as an average of the current pixel and its successor. For Chroaminance the data alternates from U to V from line to line. In order to average two like samples a delay of 2 cycles is used.

7.2.b. Vertical Filter Hardware Description

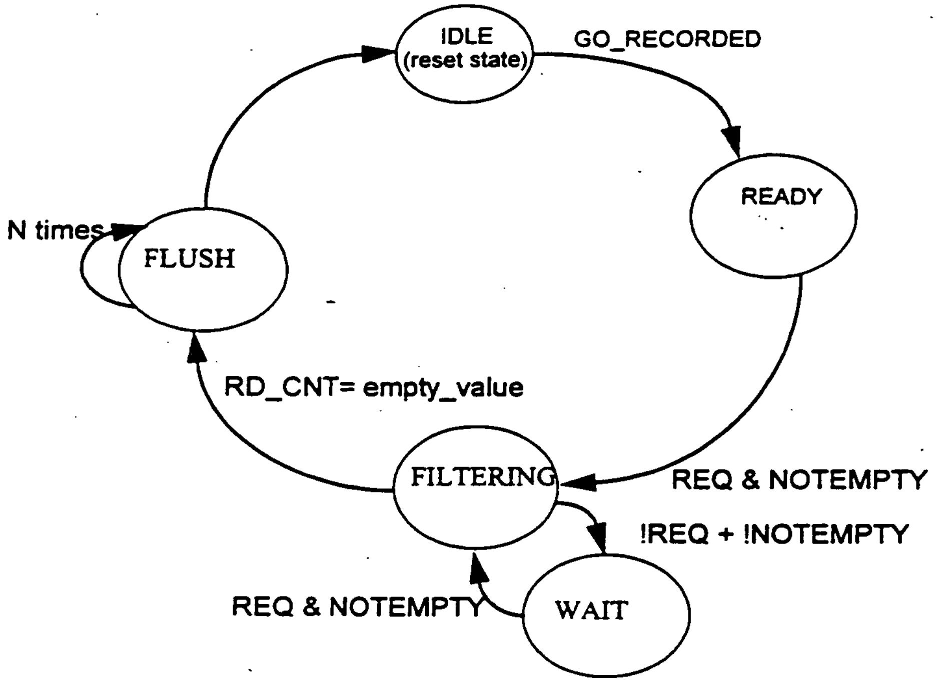
The block diagram here shows the function of the hardware used to realize the vertical filter.



(May not be necessary – depends on how chroma is handled)

There are four of these blocks used – one connected to each output of the horizontal filter section.

8. Master Control



8.1. Introduction

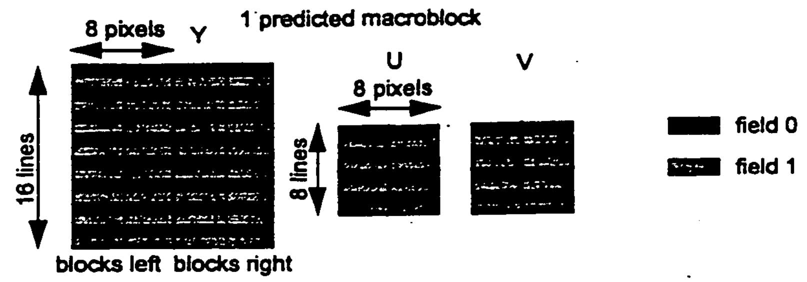
The control section maintains the data flow through the filter.

9. Output Fifos description

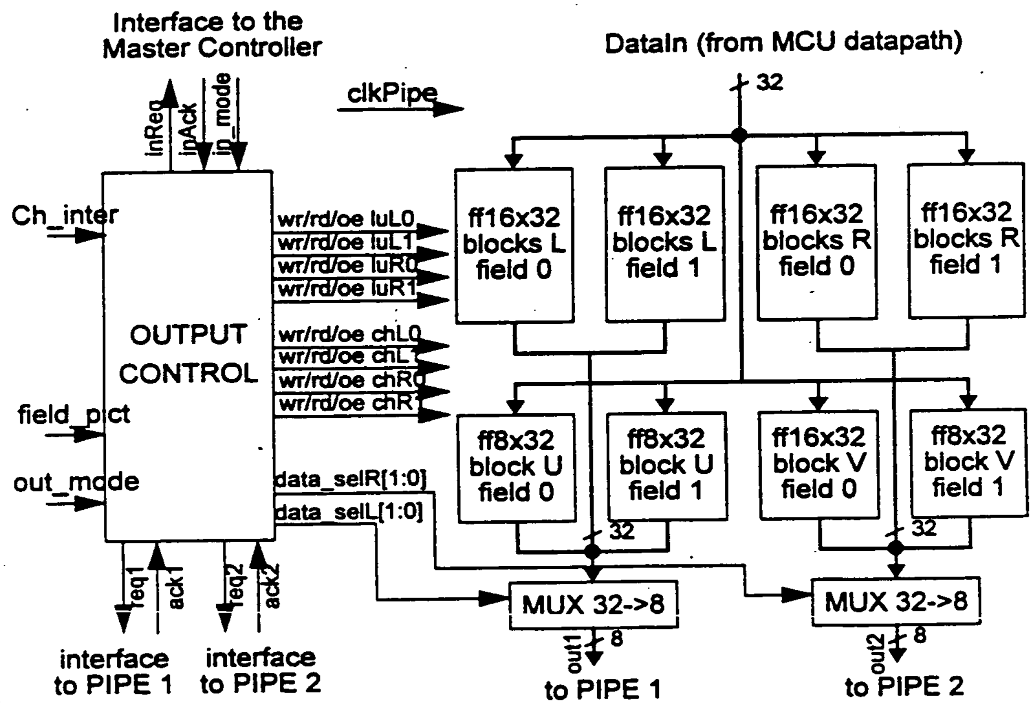
9.1. Introduction

The purpose of the output stage of the MCU is to do the interface between the prediction datapath and the adders of the pipelines. The MCU datapath generate the predicted macroblock. Luma and chroma are separated in the output fifos, as well as both fields and, in luma and chroma, left blocks

(blocks L) and right blocks (blocks R).



9.2. *Block diagram*



9.3. *Output stage interfaces*

The output stage of the MCU runs with clockPipe (54 MHz).

Interface to pipes : req1, ack1, req2, ack2, ou1, out2 : see section 4.3 of this document.

Interface to Master Controller :

- in_mode : signal is high when filtering from MCU is done in field mode (only valid if field_pict = '0')

- inReq : high when the output fifos are ready to get data from the datapath
- inAck : high when datapath is outputting correct data

Interface to datapath :

- DataIn[31:0] : data from the pipe, corresponding to 4 horizontal predicted pixels

Static interface (i.e. information that can only change between pictures)

- Ch_inter : chroma input interleaved (depends on compression type)
- field_pict : high when "field_structure" type picture
- out_mode : high when output of the MCU is of field type (only valid if field_pict = '0')

9.4. Functionality

The output fifos are composed of four ff16x32 fifos to store the luma data and four ff8x32 fifos to store the luma data.

9.4.a. Frame structure picture

There are four kind of input, depending on in_mode and Ch_inter.

In all the cases, the order inside a packet of data is always raster scan (i.e. line after line, from left to right in a line).

L prog means luma of the whole macroblock (progressive line scanning)
L f0 means luma lines of the field 0 (first field)
L f1 means luma lines of the field 1 (second field)
C prog means interleaved words of U and V for the whole macroblock (progressive line scanning)
U prog means U chroma of the whole macroblock (progressive line scanning)
V prog means V chroma of the whole macroblock (progressive line scanning)
C f0 means interleaved words of U and V for the field 0
C f1 means interleaved words of U and V for the field 1
U f0 means U chroma lines of the field 0
U f1 means U chroma lines of the field 1
V f0 means V chroma lines of the field 0
V f1 means V chroma lines of the field 1

- If in_mode is low and Ch_inter is low the order is :

- L prog (16 lines, 64 words)
- U prog (8 lines, 16 wods)
- V prog (8 lines, 16 words)

- If in_mode is low and Ch_inter is high the order is :

- L prog (16 lines, 64 words)
- C prog (8 lines, 32 words)

- If in_mode is high and Ch_inter is low the order is :

- L f0 (8 lines, 32 words)
- L f1 (8 lines, 32 words)
- U f0 (4 lines, 8 wods)
- U f1 (4 lines, 8 words)
- V f0 (4 lines, 8 wods)

- V f1 (4 lines, 8 words)

- If in_mode is high and Ch_inter is high the order is :

- L f0 (8 lines, 32 words)
- L f1 (8 lines, 32 words)
- C f0 (4 lines, 16 wods)
- C f1 (4 lines, 16 words)

The output order depens on out_mode (field or frame). There are two output connected to both pipes. A multiplexing from 32 to 8 is done. For every pipe, the output order is exactly the same as the output of the pipe "reordering fifos". See the reordering fifos document for the output order.

9.4.b. Field structure picture

in_mode has no meaning in this case : the input order is always the same. Data coming from the datapath are of field type but the scanning in the prediction is equivalent to the progressive scanning. So the input can be considered the same as above when in_mode is low.

9.4.c. Control

The req signal to the pipe is asserted when data in the output fifos are ready to be read from the pipe (taking in account the output order). When ack from the pipe is high, a new data has to be delivered immediately.

It is interesting to generate 2 internal signals : flushL and flushC that are managed this way :

- flushL rises when all the luma of a macroblock has been written to the output fifos and falls when all the luma of the macroblock has been read by the pipes.

- flushC rises when all the chroma of a macroblock has been written to the output fifos and falls when all the chroma of the macroblock has been read by the pipes.

To have a easy control of the input, inReq is simply generated as not(flushL and flushC). This ensure than never the luma of two macroblocks or the chroma of two macroblocks will share the output fifos.

HD MPEG VIDEO DECODER

APPENDIX N

INTERNAL DATA BUS

Thomson Consumer Electronics, Inc.
Indianapolis, Indiana, USA

Revision No. 2.1

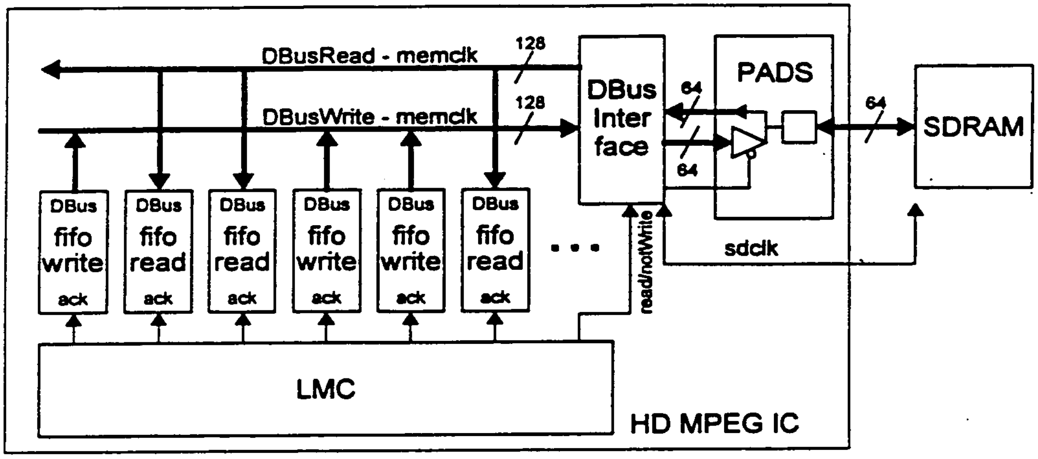
TCE PROPRIETARY AND CONFIDENTIAL

1. *Introduction*

The HDMPEG IC has two internal 128 memory busses. One bus is used by the fifos that write into the memory, the other by the fifos that read the memory.
This document describes the dbus interface block, which is the interface between these busses and the external SDRAM data bus. It also describes the interfaces between the internal busses and the fifos.

2. *Global description*

Schematic of the SDRAM data interface



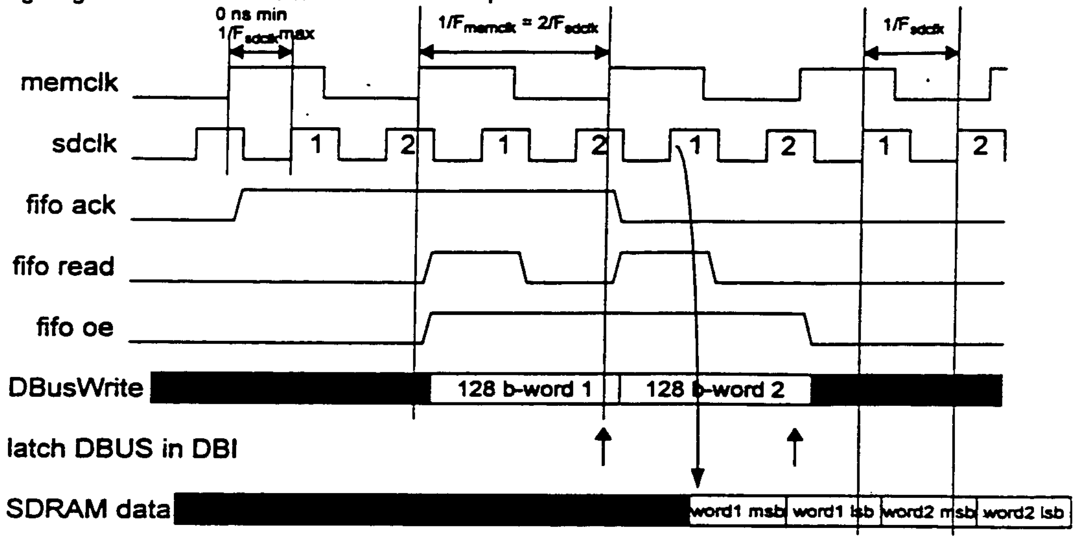
The fifos that send data to the SDRAM are connected to the 128-bit DBusWrite bus.
The fifos that receive data from the SDRAM are connected to the 128-bit DBusRead bus.
Every fifo controller receive an acknowledge signal from the LMC to generate the write or read pulses of the fifo it controls. For the write fifos, the acknowledge signal is also used to get the drive on the bus.
The DBus interface function is to multiplex/demultiplex between the internal and external busses and to latch data from or to the pads at the high speed SDRAM clock (sdclk) rate (dobble of the memory clock (memclk) rate).

3. *SDRAM accesses*

For the SDRAM clock, the first rising edge after internal memclk rising edge is defined as rising edge "1". The other is the rising edge "2".
CAS access are output out to the chip on sdclk rising edge "1" and strobe on the SDRAM on sdclk rising edge "2".

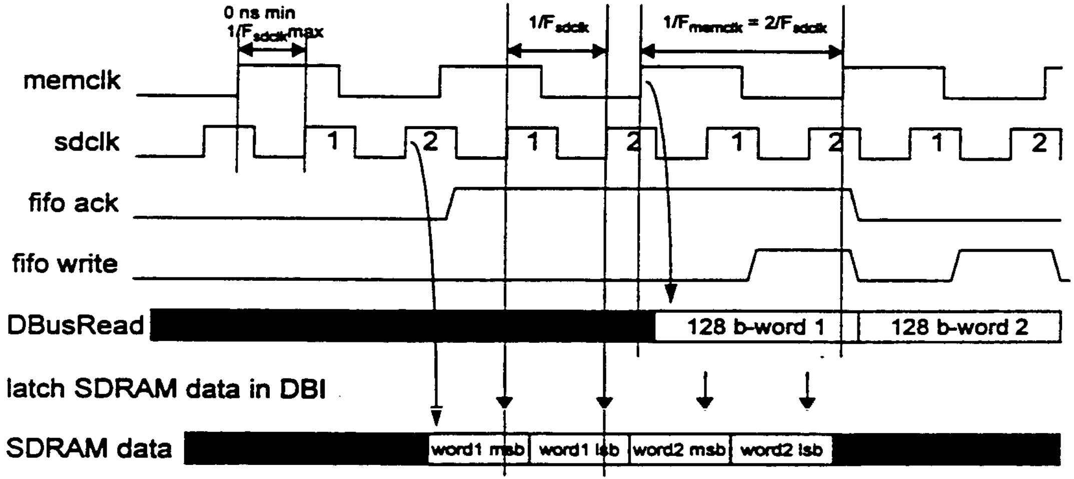
3.1. *Write access*

Timing diagram with two 128-bits word write example



3.2. *Read access*

CAS access are output of the chip on sdclk pulse "1" and strobe on the SDRAM on sdclk pulse "2".
Timing diagram with two 128-bits word read example (CAS latency = 3)

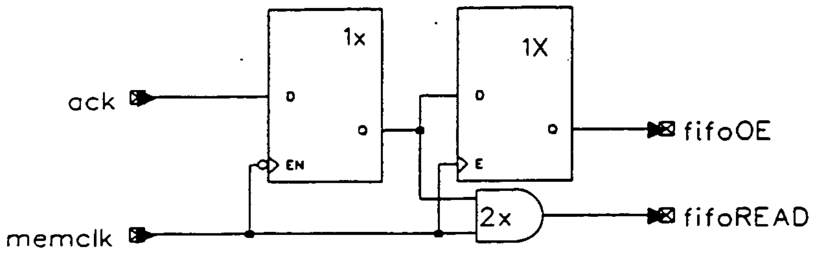


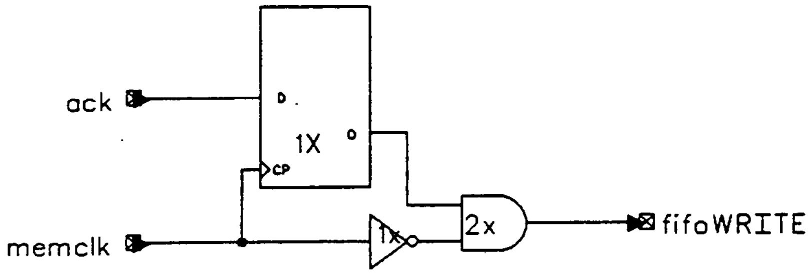
4. *Fifo interface*

4.1. *Generation of fifo read signal from ack*

4.2. *Generation of fifo write signal from ack*

Buffering of fifoREAD and fifoWRITE must be adapted to the specified fifo or group of fifos.





5. *DBus interface specification*

INPUTS :

- DBusWrite[127:0]
- DDin[63:0]
- sdclk
- memclk
- sdramReadNotWrite

OUTPUTS

- DBusRead[127:0]
- DDout[63:0]

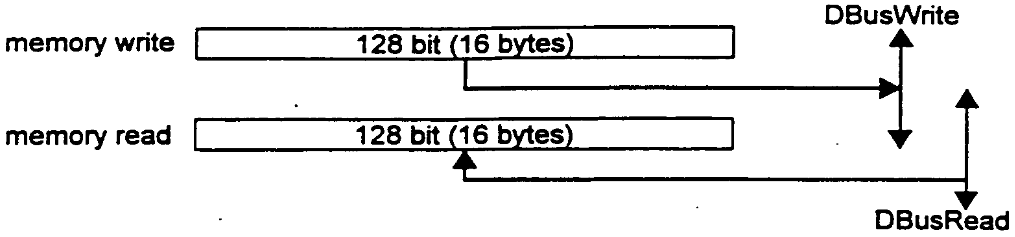
Internal sdclk is derived from the external one through an analog pad.

6. Dbus FIFO size

Fifo size are limited to 32-bit words. So 128-bit fifos are splitted in 4 fifos in parallel.
General naming of fifo use the following convention : ffwxb where w is the number of word of the fifo and b the number of bit per word.

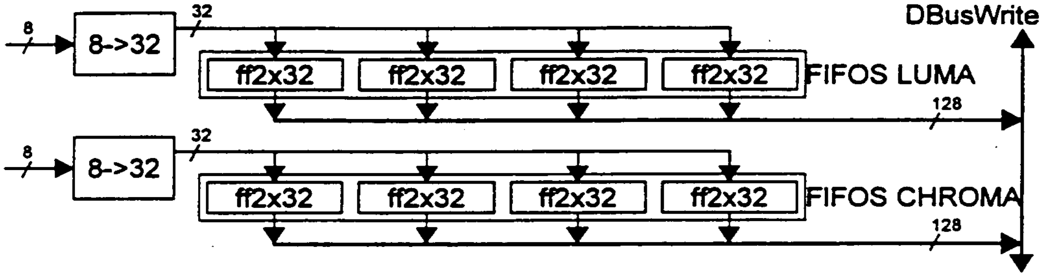
6.1. Memory interface write, read and block copy

These fifos are done with latches. Each is one 128-bit word.



6.2. Standard definition video input, luma and chroma

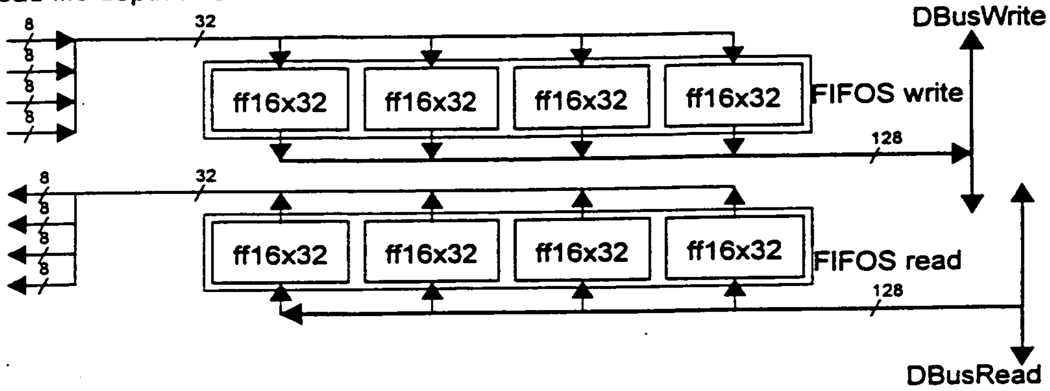
- luma fifo depth = two 128-bit words
- chroma fifo depth = two 128-bit words



6.3. LMU write to and read from the memory

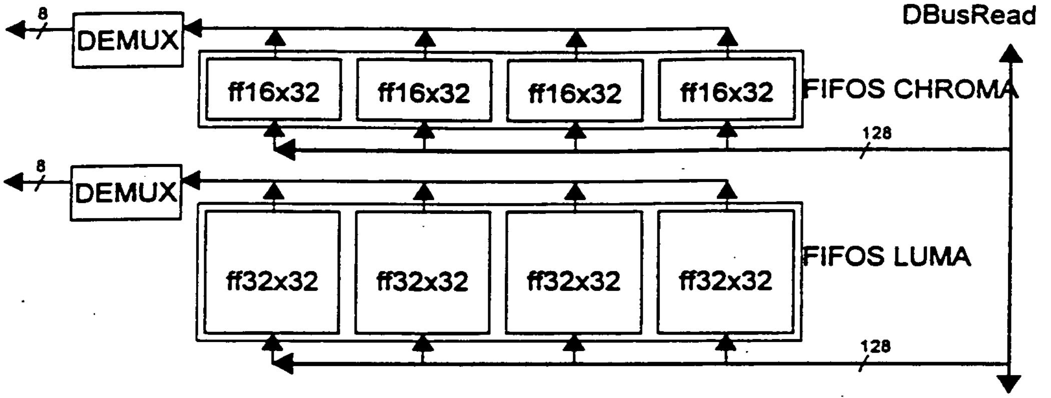
- LMU write fifo depth : 16 128-bit words

- LMU read fifo depth : 16 128-bit words



6.4. Display for block to line conversion or decompression

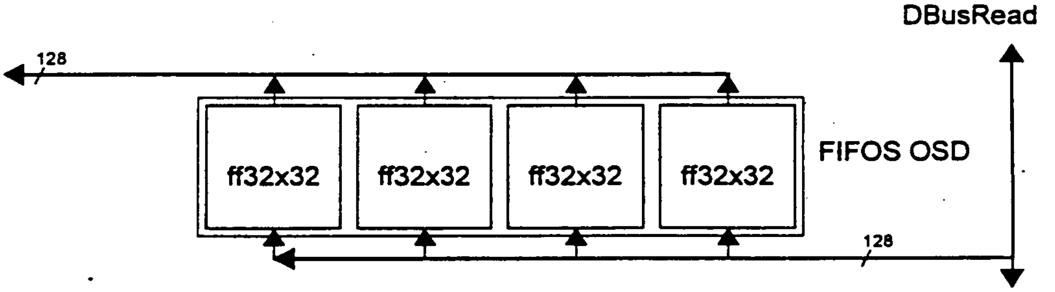
- Chroma fifo depth : 16 128-bits words
- Luma fifo depth : 32 128-bits words



In no compression and H/2-M/2 modes, luma and chroma accesses are separated. Two requests and two controls are generated.
In 2M/3 mode, access is done for both during the same burst : three 128-bits words are sent to the luma fifos then one 128-bits word is sent to the chroma fifos, this 4 times. One single request is generated (signal lumaReq is used).

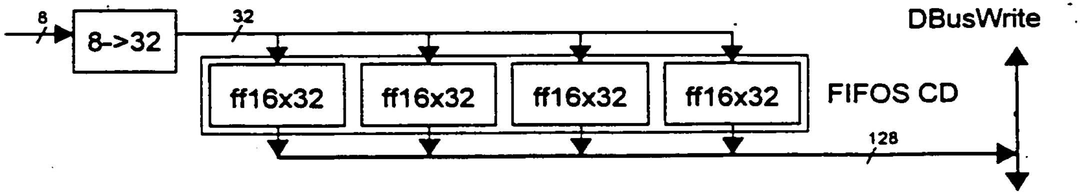
6.5. OSD

- OSD fifo depth : 32 128-bits words28-bits words



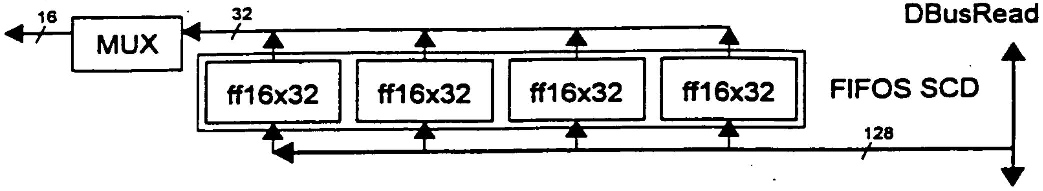
6.6. Compressed data write

- CD fifo depth : 16 128-bits words



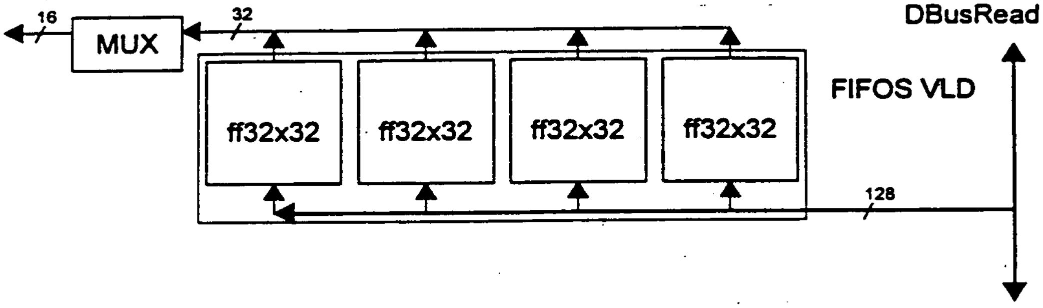
6.7. *Compressed data read for Start code detector*

- CD fifo depth : 16 128-bits words



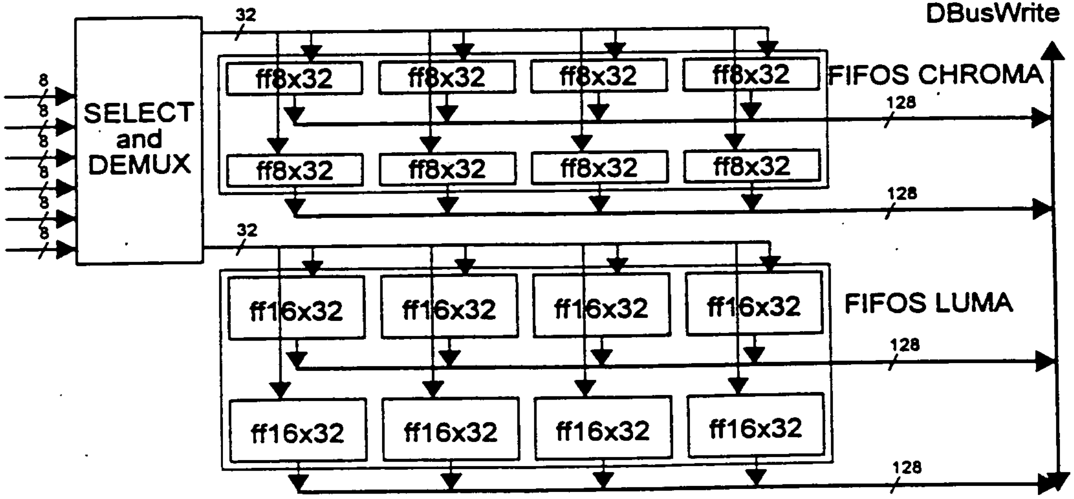
6.8. *Compressed data read for VLD*

- CD fifo depth : 32 128-bits words



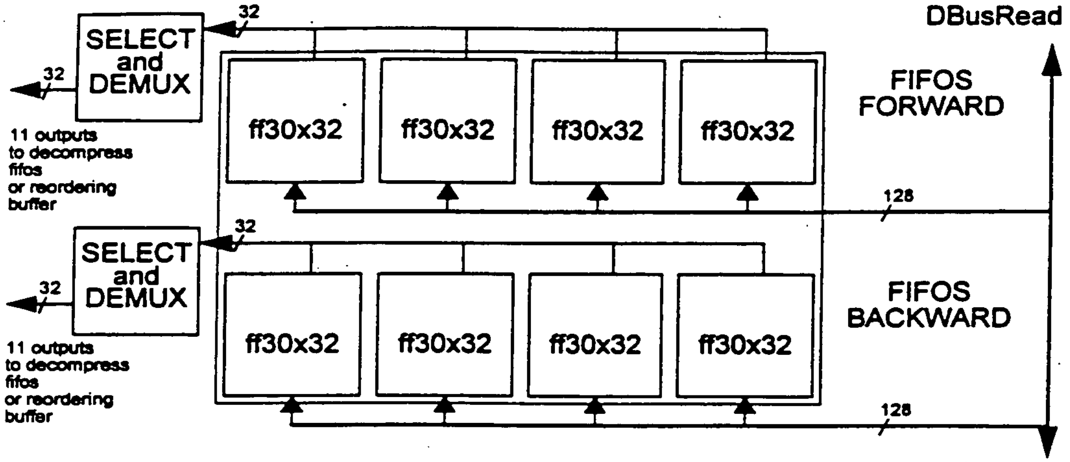
6.9. *Reconstruction macroblock*

- Chroma fifos depth : 16 128-bits words
- Luma fifos depth : 32 128-bits words



6.10. Prediction macroblock

- forward or backward fifos depth : 30 128-bits words logically



HD MPEG VIDEO DECODER

APPENDIX O

DISPLAY FIFO DESIGN

Thomson Consumer Electronics, Inc.
Indianapolis, Indiana, USA

Revision No. 2.1

TCE PROPRIETARY AND CONFIDENTIAL

1. Introduction

This document describes the function of the HD MPEG IC reordering fifos and the interfaces between the IDCT, the adder to the compression unit, and the reordering fifos. Figure A shows a block diagram of the reordering fifos section. Note that this block is used twice in the HD MPEG IC, once for each pipe.

2. Description of interfaces

2.1. Pipe controller to reordering fifos interface

The pipe controller stops and starts the reordering fifos as well as the pipe. This is accomplished with the control signals fifos_active, fill, and flush. Two other signals, in_field and out_field control the format conversion of the reordering fifos.
The interface signals for this block are:

fifos_active - active high signal indicating the reordering fifos should accept and produce data on the next clock cycle.

fill - Active high signal indicating the reordering fifos should accept data on the next clock cycle..

flush - Active high signal indicating the reordering fifos should produce data on the next clock cycle.

in_field - High indicates the input data from the IDCT block is in a field format. Low indicates the input data from the IDCT block is in a frame format.

out_field - High indicates the output data to the compression block should be in a field format. Low indicates the output data to the compression block should be in a frame format.

2.2. IDCT to reordering fifos interface

Each IDCT block outputs a 9 bit word (2's compliment pixel) per cycle at 54MHz. This interface must not stop during normal operation. The pixel order is input in columns.
The interface signals for this block are:

idct_bus(8:0) - 9 bit 2's compliment pixel from the IDCT to the ordering fifos.

The ordering of the pixel data is a half macro block of two Y and one C block. Each 8 x 8 block of Y or C pixels is input prior to starting the next block. Each 8 pixel column starting on the left of the block is input prior to starting the next colum. Each pixel of an 8 pixel column is input starting from the top of each column. Figures E and F show a graphical description of this pixel ordering. Note, there is no functional difference between the way Y and C pixels are handled at the input to the reordering fifos.

2.3. Reordering fifos interface to the adder to the compression unit

Each reordering fifo outputs a 9 bit word (2's compliment pixel) per cycle at 54MHz. This interface should not

stop during normal operation. The pixel order is output in rows.
The interface signals for this block are:

comp_bus(8:0) - 9 bit 2's compliment pixel from the ordering fifos to adder.

The ordering of the pixel data is a half macro block of two Y and one C block. The two 8 x 8 Y blocks are output together in an interleaved format, i.e. a pixel from Y1, then Y3, then Y1, Y3 again, and so on. Each 8 pixel row starting at the top of a block is ouput prior to startiing the next row. Each pixel of an 8 pixel row is output starting from the left of each row before starting another row. C blocks are handled differently. Each 8 x 8 C block is treated as four 4 x 4 blocks. Two of these blocks (1 and 2, then 3 and 4) are read out top to bottom, left to right as with Y blocks. Figures E and F show a graphical representaion of this pixel ordering.

3. *Reordering fifos functionality*

The purpose of the reordering fifos is to receive pixel data from the IDCT in either a field or frame format and output the data to the adder prior to compression in either a frame or field format. Each macro block may be formated in either manner independent of other macro-blocks. Figure E shows a graphical representation of a frame to field conversion. Figure F shows a field to frame conversion. No conversion, frame input to frame output or field input to field output, is also possible. See the LMC specification for additional information on frame and field formats.

The computation itself is split into two pipes for bandwidth purposes. Figure A shows a block diagram of the reordering fifos in a single pipe. Each pipe handles two Y blocks and one C block from each macro block. The upper pipe handles blocks 1 and 3 of Y and one block, U, of chroma. The lower pipe handles blocks 2 and 4 of Y and the one block, V, of chroma. Although, the two pipes use the same 54MHz clock the data within the two pipes is not strictly synchronized. In other words, one cannot be certain that X number of clock cycles after the first pipe finishes block Y 1 the lower pipe will finish block Y 3. An upper bound of X <= 96?? and a lower bound of X >= 64?? can be calculated for normal operation.

The synchronization between the reordering fifos output (i.e. the pipe controller), the motion compensation unit, and the compression unit will use a request acknowledge structure. The compression unit will generate a request signal for each word of data. The pipe controller and motion compensation unit must both acknowledge before the data may be considered valid. During normal operation these actions will occur every clock cycle. Initialization (filling the pipeline) and flushing the pipeline are two exceptions to normal operation. See the pipe controller specification for more details.

Figure E: Reordering fifos frame to field conversion

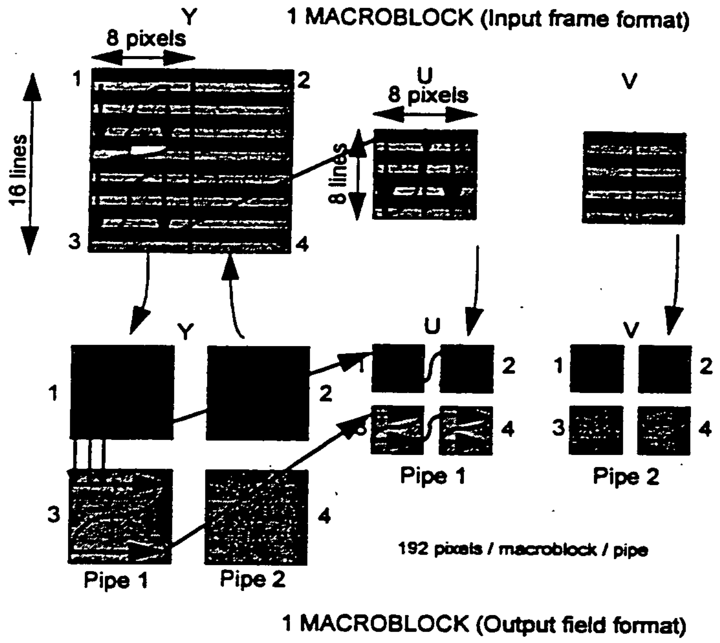
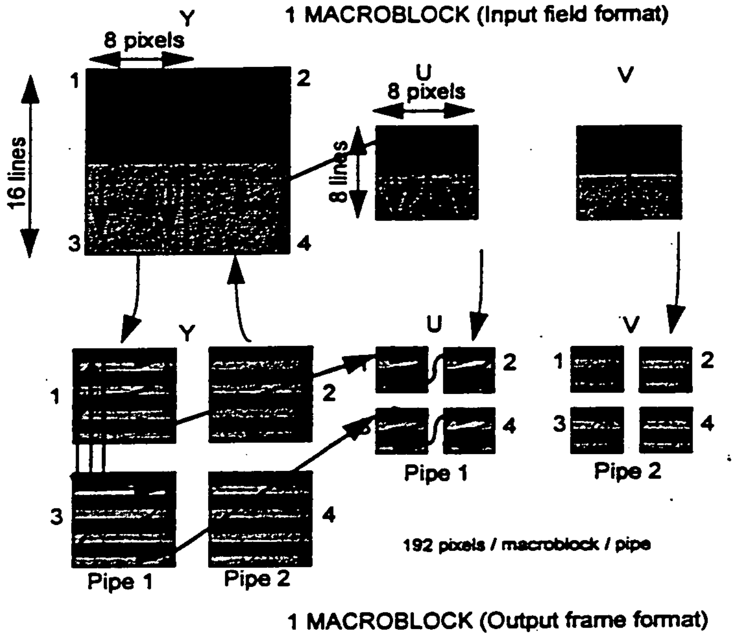
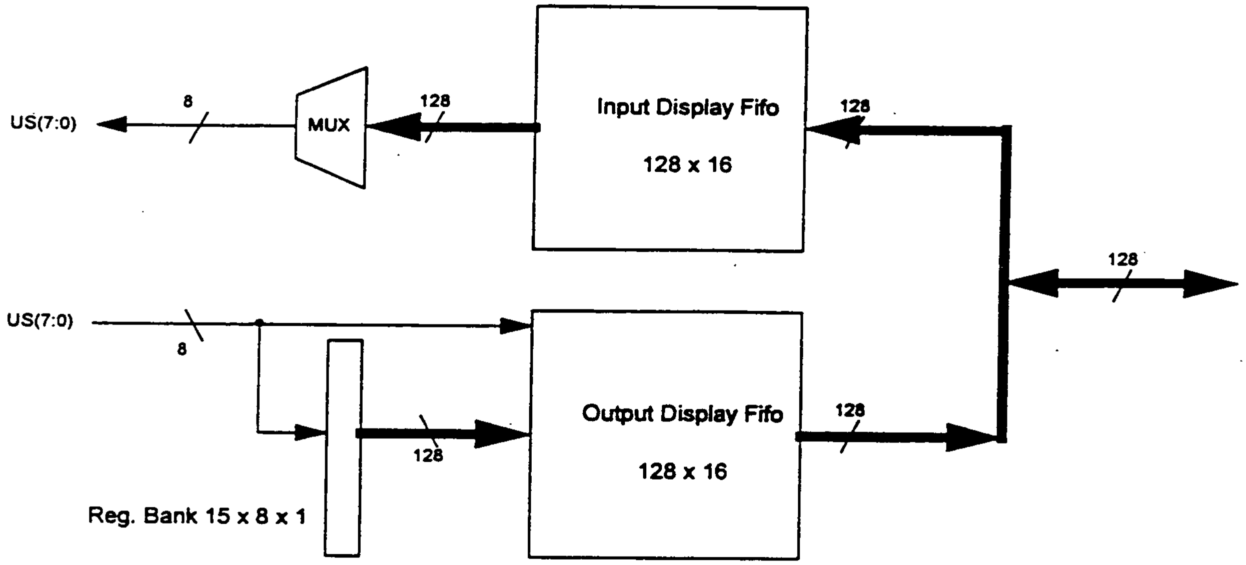

Y
1 MACROBLOCK (Input frame format)
8 pixels
16 lines
1
2
3
4
U
8 pixels
8 lines
V
Y
1
2
3
4
Pipe 1
Pipe 2
U
1
2
3
4
Pipe 1
V
1
2
3
4
Pipe 2
192 pixels / macroblock / pipe
1 MACROBLOCK (Output field format)

Figure F: Reordering fifos field to frame


Y
1 MACROBLOCK (Input field format)
8 pixels
1
2
16 lines
3
4
U
8 pixels
8 lines
V
Y
1
2
3
4
Pipe 1
Pipe 2
U
2
4
Pipe 1
V
1
2
3
4
Pipe 2
192 pixels / macroblock / pipe
1 MACROBLOCK (Output frame format)

A: Display FIFOs Block Diagram



HD MPEG VIDEO DECODER

APPENDIX P

REORDERING FIFO DESIGN

Thomson Consumer Electronics, Inc.
Indianapolis, Indiana, USA

Revision No. 2.1

THESE DRAWINGS AND SPECIFICATIONS ARE THE PROPERTY OF THOMSON CONSUMER ELECTRONICS, INC. AND SHALL NOT BE REPRODUCED OR COPIED OR USED AS THE BASIS FOR THE MANUFACTURE OR SALE OF APPARATUS OR DEVICES WITHOUT WRITTEN PERMISSION.

TCE PROPRIETARY AND CONFIDENTIAL

1. Introduction

This document describes the function of the HD MPEG IC reordering fifos and the interfaces between the IDCT, the adder to the compression unit, and the reordering fifos. Figure A shows a block diagram of the reordering fifos section. Note that this block is used twice in the HD MPEG IC, once for each pipe.

2. Description of interfaces

2.1. Pipe controller to reordering fifos interface

The pipe controller stops and starts the reordering fifos as well as the pipe. This is accomplished with the control signals fifos_active, fill, and flush. Two other signals, in_field and out_field control the format conversion of the reordering fifos.

The interface signals for this block are:

fifos_active - active high signal indicating the reordering fifos should accept and produce data on the next clock cycle.

fill - Active high signal indicating the reordering fifos should accept data on the next clock cycle..

flush - Active high signal indicating the reordering fifos should produce data on the next clock cycle.

in_field - High indicates the input data from the IDCT block is in a field format. Low indicates the input data from the IDCT block is in a frame format.

out_field - High indicates the output data to the compression block should be in a field format. Low indicates the output data to the compression block should be in a frame format.

2.2. IDCT to reordering fifos interface

Each IDCT block outputs a 9 bit word (2's compliment pixel) per cycle at 54MHz. This interface must not stop during normal operation. The pixel order is input in columns.

The interface signals for this block are:

idct_bus(8:0) - 9 bit 2's compliment pixel from the IDCT to the ordering fifos.

The ordering of the pixel data is a half macro block of two Y and one C block. Each 8 x 8 block of Y or C pixels is input prior to starting the next block. Each 8 pixel column starting on the left of the block is input prior to starting the next colum. Each pixel of an 8 pixel column is input starting from the top of each column. Figures E and F show a graphical description of this pixel ordering. Note, there is no functional difference between the way Y and C pixels are handled at the input to the reordering fifos.

2.3. Reordering fifos interface to the adder to the compression unit

Each reordering fifo outputs a 9 bit word (2's compliment pixel) per cycle at 54MHz. This interface should not stop during normal operation. The pixel order is output in rows.

The interface signals for this block are:

comp_bus(8:0) - 9 bit 2's compliment pixel from the ordering fifos to adder.

The ordering of the pixel data is a half macro block of two Y and one C block. The two 8 x 8 Y blocks are output together in an interleaved format, i.e. a pixel from Y1, then Y3, then Y1, Y3 again, and so on. Each 8 pixel row starting at the top of a block is ouput prior to startiing the next row. Each pixel of an 8 pixel row is output starting from the left of each row before starting another row. C blocks are handled differently. Each 8 x 8 C block is treated as four 4 x 4 blocks. Two of these blocks (1 and 2, then 3 and 4) are read out top to bottom, left to right as with Y blocks. Figures E and F show a graphical representaion of this pixel ordering.

3. *Reordering fifos functionality*

The purpose of the reordering fifos is to receive pixel data from the IDCT in either a field or frame format and output the data to the adder prior to compression in either a frame or field format. Each macro block may be formated in either manner independent of other macro-blocks. Figure E shows a graphical representation of a frame to field conversion. Figure F shows a field to frame conversion. No conversion, frame input to frame output or field input to field output, is also possible. See the LMC specification for additional information on frame and field formats.

The computation itself is split into two pipes for bandwidth purposes. Figure A shows a block diagram of the reordering fifos in a single pipe. Each pipe handles two Y blocks and one C block from each macro block. The upper pipe handles blocks 1 and 3 of Y and one block, U, of chroma. The lower pipe handles blocks 2 and 4 of Y and the one block, V, of chroma. Although, the two pipes use the same 54MHz clock the data within the two pipes is not strictly synchronized. In other words, one cannot be certain that X number of clock cycles after the first pipe finishes block Y 1 the lower pipe will finish block Y 3. An upper bound of X <= 96?? and a lower bound of X >= 64?? can be calculated for normal operation.

The synchronization between the reordering fifos output (i.e. the pipe controller), the motion compensation unit, and the compression unit will use a request acknowledge structure. The compression unit will generate a request signal for each word of data. The pipe controller and motion compensation unit must both acknowledge before the data may be considered valid. During normal operation these actions will occur every clock cycle. Initialization (filling the pipeline) and flushing the pipeline are two exceptions to normal operation. See the pipe controller specification for more details.

Figure E: Reordering fifos frame to field conversion

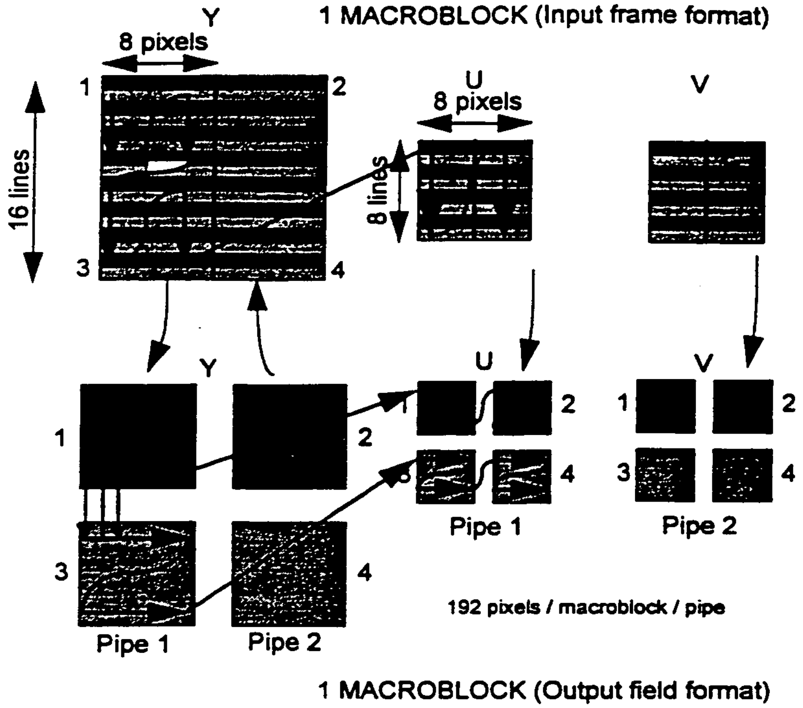
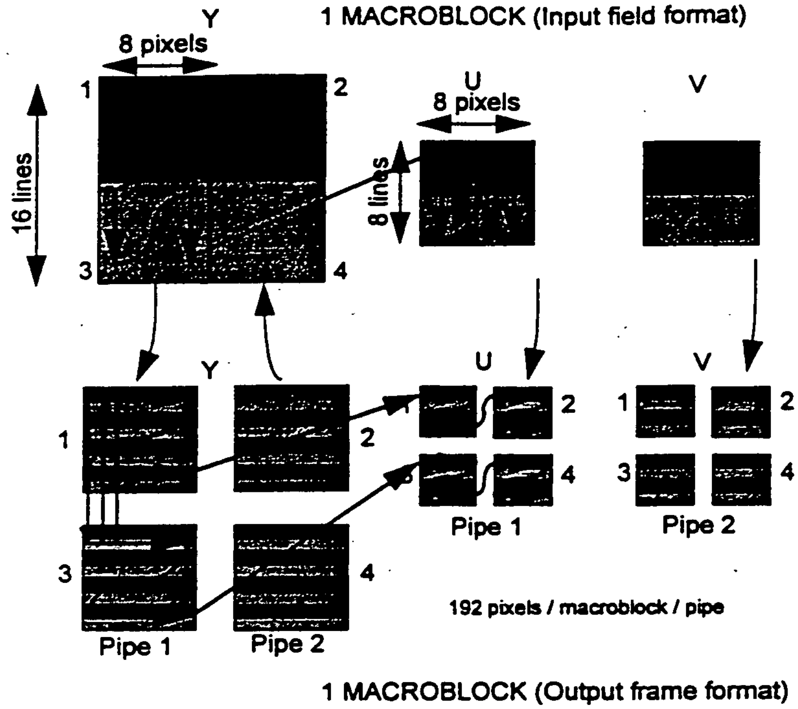

Y
1 MACROBLOCK (Input frame format)
8 pixels
1
2
16 lines
3
4
U
8 pixels
8 lines
V
Y
1
2
3
4
Pipe 1
Pipe 2
U
1
2
3
4
Pipe 1
V
1
2
3
4
Pipe 2
192 pixels / macroblock / pipe
1 MACROBLOCK (Output field format)

Figure F: Reordering fifos field to frame


Y
1 MACROBLOCK (Input field format)
8 pixels
16 lines
1
2
3
4
U
8 pixels
8 lines
V
Y
1
2
3
4
Pipe 1
Pipe 2
U
2
4
Pipe 1
V
1
2
3
4
Pipe 2
192 pixels / macroblock / pipe
1 MACROBLOCK (Output frame format)

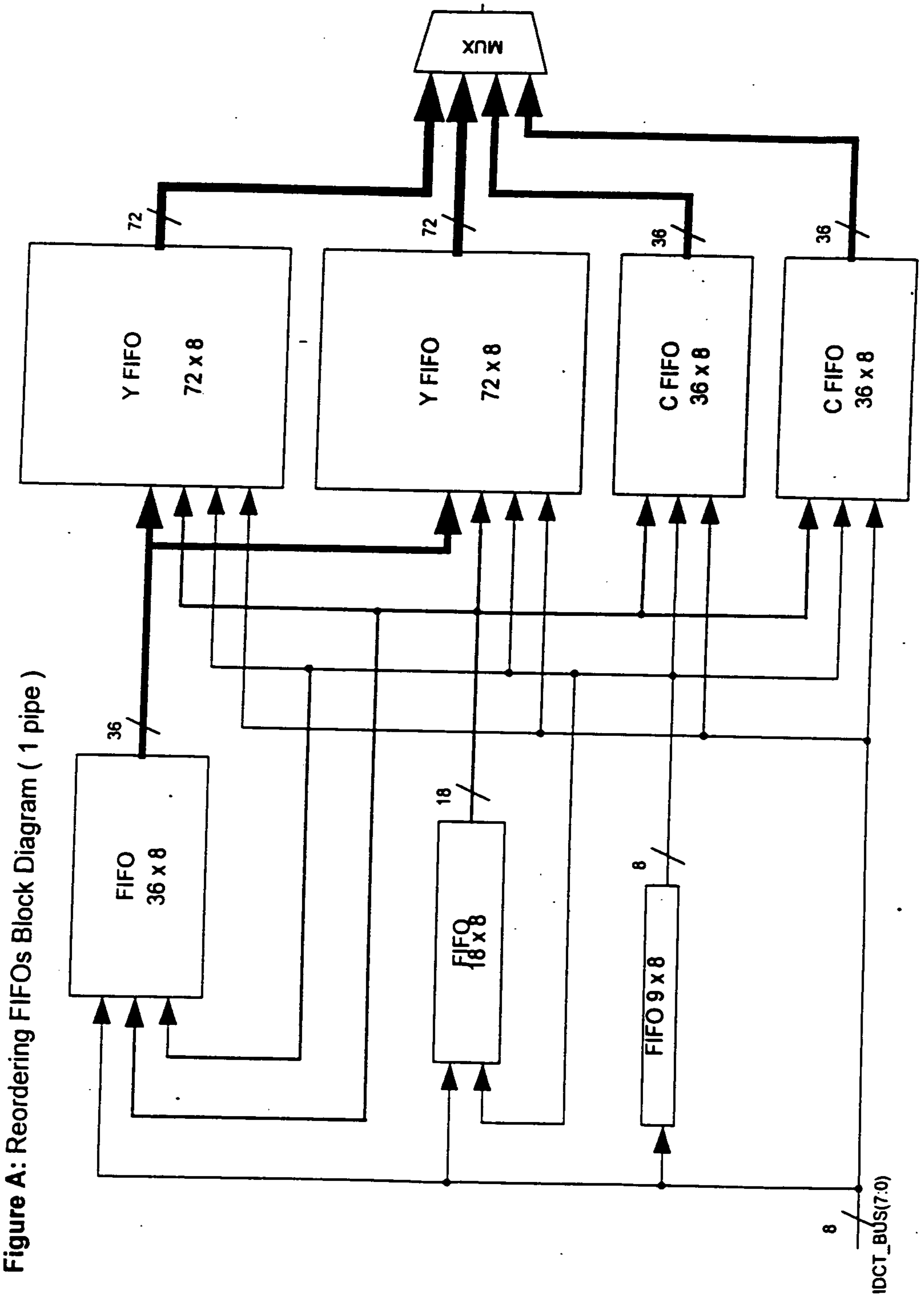


Figure A: Reordering FIFOs Block Diagram (1 pipe)

HD MPEG VIDEO DECODER

APPENDIX Q

RECONSTRUCTION FIFO DESIGN

Thomson Consumer Electronics, Inc.
Indianapolis, Indiana, USA

Revision No. 2.1

THESE DRAWINGS AND SPECIFICATIONS ARE THE PROPERTY OF THOMSON CONSUMER ELECTRONICS, INC. AND SHALL NOT BE REPRODUCED OR COPIED OR USED AS THE BASIS FOR THE MANUFACTURE OR SALE OF APPARATUS OR DEVICES WITHOUT WRITTEN PERMISSION.

TCE PROPRIETARY AND CONFIDENTIAL

1. *Introduction*

This document describes the function of the HD MPEG IC reconstruction fifos and the interfaces between the compression block, the LMC and the reconstruction fifos. Figure A shows a block diagram of the reconstruction fifos section.

2. *Description of interfaces*

2.1. *Control Registers*

The reconstruction fifos use a 2 bit register to indicate the compression mode being used. This register is loaded via the application bus and appears as static data to the reconstruction fifos.

The interface signals for this block are:

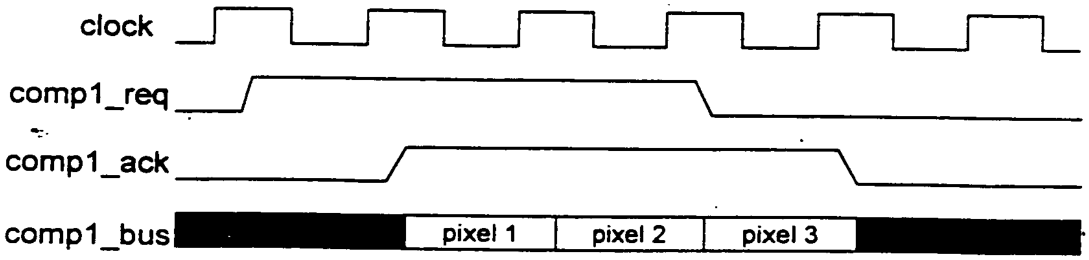
comp_reg(1:0) - 2 bit register indicating the compression mode being used. A 00 indicates normal mode. No compression is used, and the compression unit is bypassed. A 01 indicates 2M/3 mode or 33% compression. Finally, a 10 indicates M/2, H/2 mode or 75% compression.

2.2. *Compression to reconstruction fifos interface*

Each compression block outputs 8 bits (1 pixel) per cycle at 54MHz. This interface must not stop during normal operation. The pixel order is input in rows.

The exchange of data is initiated by a request from the fifo. The compression unit responds with an acknowledge and a valid pixel of data. Figure B shows a timing diagram of this interface.

Figure B: Timing diagram IDCT to reordering fifos



The interface signals for this block are:

comp1_bus(7:0) - 8 bit pixel from the pipe 1 compression unit to the reconstruction fifos.

comp1_req - Active high request signal from the reconstruction fifos to the pipe 1 compression unit.

comp1_ack - Active high acknowledge signal from the pipe 1 compression unit to the reconstruction fifos.

comp2_bus(7:0) - 8 bit pixel from the pipe 2 compression unit to the reconstruction fifos.

comp2_req - Active high request signal from the reconstruction fifos to the pipe 2 compression unit.

comp2_ack - Active high acknowledge signal from the pipe 2 compression unit to the reconstruction fifos.

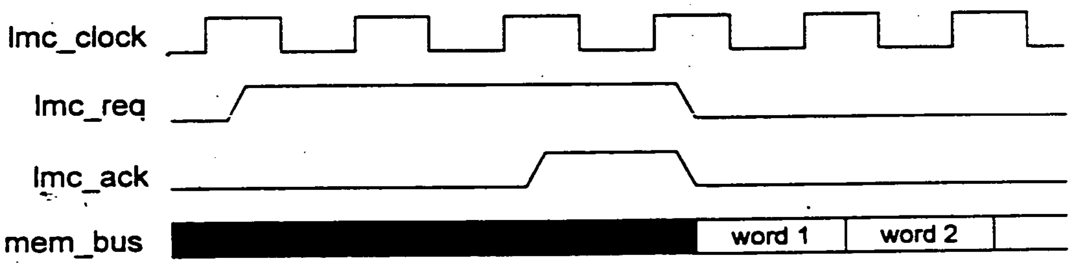
The input of the pixel data to the reconstruction fifos consists of two pipes each delivering a half macro block of two Y and one C blocks. The number of pixels per row and number of rows per block varies depending on the compression mode. The two 8 x 8 Y blocks are received together interleaved on a pixel basis, i.e. a pixel from Y1, then Y3, then Y1, and Y3 again, and so on. In the case of Y blocks, each row of pixels starting from the top of a Y block is input prior to starting the next row. The pixels in each row are input starting from the left side of the row. Each C block is treated as four C blocks. A pair of these blocks is read out top to bottom, left to right as with Y blocks. The four ; blocks themselves are received in the pairs: 1 and 3, 2 and 4. Figure D shows a graphical repre-ntaion of this pixel ordering during normal mode (no compression). Figure E shows a graphical representation of this pixel ordering during 2M/3 mode (33% compression). Figure F shows a graphical representation of this pixel ordering during M/2 mode (75% compression).

2.3. *Reconstruction Fifos to LMC*

This interface outputs 128 bit words over the memory bus in a burst format.

The exchange of data is initiated by a request from the reconstruction fifos. When available the LMC responds with an acknowledge. The reconstruction unit returns the request line to an inactive status and a valid word of data is placed on the memory bus. Additional words are placed on the memory bus on consecutive cycles until the entire burst write is completed. The number of words in each burst write is dependent on the mode. See section 3.0 of this document for the number of words written during each mode. Figure C shows a timing diagram of this interface.

Figure C: Timing diagram from reconstruction fifos the memory



The interface signals for this block are:

lmc_clock - 54 MHz clock synchronous to the LMC.

mem_bus(127:0) -128 bit (16 pixel) word from the reordering fifos to the LMC.

lmc_req - Active high burst write request from the reconstruction fifos to the LMC controller.

lmc_ack - Active high burst enable from the LMC to the reconstruction fifos.

3. Reconstruction fifos functionality

The purpose of the reconstruction fifos is to receive data from the two decode pipes and reconstruct the original macro block format. The data is received from the two compression units, buffered, and written to memory.

Two pixels are received per clock cycle (one from each pipe) at a clock rate of 54MHz. Figures D, E, and F show graphical representations of the order in which the pixels are received from the compression blocks. Although, the two pipes use the same 54MHz clock the data within the two pipes is not strictly synchronized. In other words, one cannot be certain that X number of clock cycles after the first pipe finishes block Y1 the lower pipe will finish block Y3. An upper bound of X <= 96?? and a lower bound of X >= 64?? can be calculated for all modes of operation. The reconstruction fifos buffer the data from each pipe prior to a memory write long enough to insure valid data is available.

Data is written to memory in blocks of 128 bit words synchronous to the LMC clock. Note that the external bus is only 64 bits per word, so two memory words correspond to one reconstruction block word. The number of words written per memory access and the arrangement of pixels in each word varies according to the mode of operation.

3.1. No compression mode write

The no compression mode is divided into luma and chroma writes. The chroma write for a particular macro block always follows the luma write. Other memory bus operations not involving the reconstruction block may occurr between luma and chroma writes. During the no compression mode luma writes consist of 24 words containing 16 pixels each. Each block is written out prior to starting another block. The blocks are written in numerical order one through four. Each block is written out starting from the top with two lines concatenated (1st line - msb, 2nd line - lsb) to produce a single word. Figure D shows a graphical representation of this pixel ordering. Table 1 lists the contents of each word during a luma write.

During normal operation (no compression), chroma writes consist of 8 words containing 16 pixels each. Each word consists entirely of 16 pixels of U or 16 pixels of V. Consecutive words alternate between paired rows of U and V. Figure D shows a graphical representation of this. Table 2 lists the contents of each word during a chroma write.

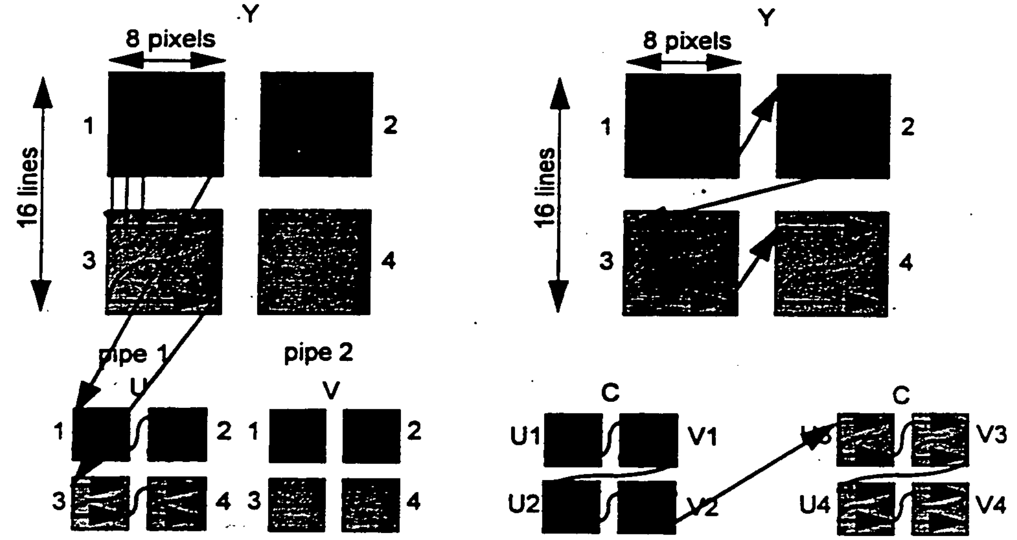
Table 1: Luma write no compression mode

Word : Contents	Word : Contents	Word : Contents	Word : Contents
1 : Y1	5 : Y2	9 : Y3	13 : Y4
2 : Y1	6 : Y2	10 : Y3	14 : Y4
3 : Y1	7 : Y2	11 : Y3	15 : Y4
4 : Y1	8 : Y2	12 : Y3	16 : Y4

Table 2: Chroma write no compression mode

Word : Contents	Word : Contents	Word : Contents	Word : Contents
1 : U1	3 : U2	5 : U3	7 : U4
2 : U1	4 : U2	6 : U3	8 : U4

Figure D: Reconstruction fifos normal mode (no compression)



384 (8 bit pixels) / macroblock

1 MACROBLOCK (Input Pixel Order)

24 (128 bit words) / macroblock

1 MACROBLOCK (Output Pixel Order)

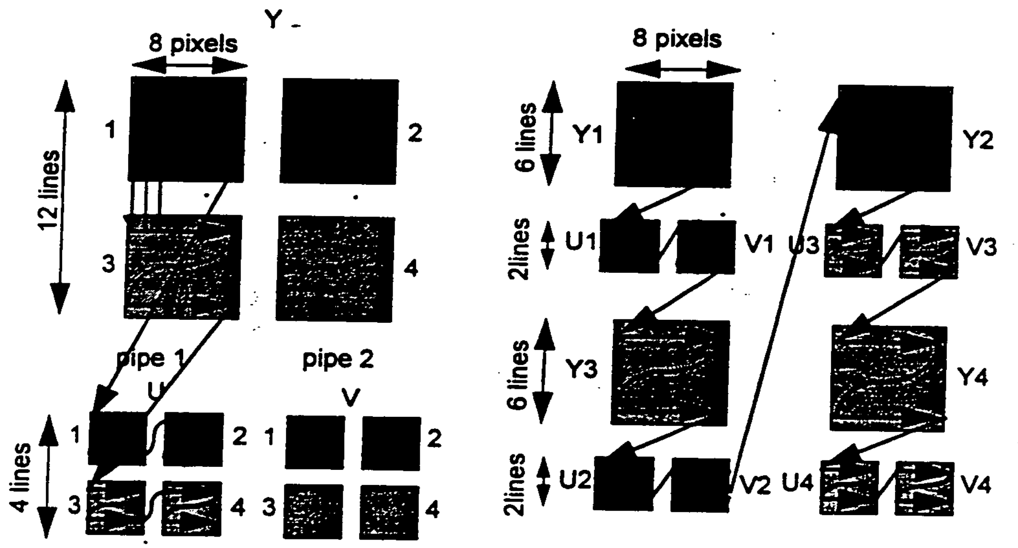
2. *2M/3 (33% compression)*

The 2M/3 mode write consist of 24 words containing 16 compressed pixels each. Each block is written out prior to starting another block. The corresponding luma and chroma pixels are interleaved in the same block with all the luma pixels followed by all the chroma pixels. Corresponding U and V chroma pixels are further concatenated into the same words with all the U pixels of a particular block as the msb followed by all of the V pixels. The blocks are written in numerical order one through four. Each block is written out starting from the top with two lines concatenated (1st line - msb, 2nd line - lsb) to produce a single word. Figure E shows a graphical representation of this pixel ordering. Table 2 lists the contents of each word during a write.

Table 3: Write 2M/3 mode (33% Compression)

Word : Contents	Word : Contents	Word : Contents	Word : Contents
1 : Y1	5 : Y2	9 : Y3	13 : Y4
2 : Y1	6 : Y2	10 : Y3	14 : Y4
3 : Y1	7 : Y2	11 : Y3	15 : Y4
4 : U1, V1(lsb)	8 : U2, V2	12 : U3, V3	16 : U4, V4

Figure E: Reconstruction fifos 2M/3 mode (33% compression)



256 (8 bit pixels) / macroblock

16 (128 bit words) / macroblock

1 MACROBLOCK (Input Pixel Order)

1 MACROBLOCK (Output Pixel Order)

3.3. *M/2 mode (75% compression)*

The M/2 mode is divided into luma and chroma writes. The chroma write for a particular macro block always follows the luma write. Other memory bus operations not involving the reconstruction block may occur between luma and chroma writes. During the M/2 mode luma writes consist of 4 words containing 16 compressed pixels each. Each block is written out prior to starting another block. The blocks are written in numerical order one through four. Each block is written out starting from the top with two lines concatenated (1st line - msb, 2nd line - lsb) to produce a single word. Figure F shows a graphical representation of this pixel ordering. Table 4 lists the contents of each word during a luma write.

During M/2 mode, chroma writes consist of 2 words containing 16 compressed pixels each. Each word consists of two consecutive pairs of 8 U pixels and 8 paired V pixels. Figure F shows a graphical representation of this. Table 5 lists the contents of each word during a chroma write.

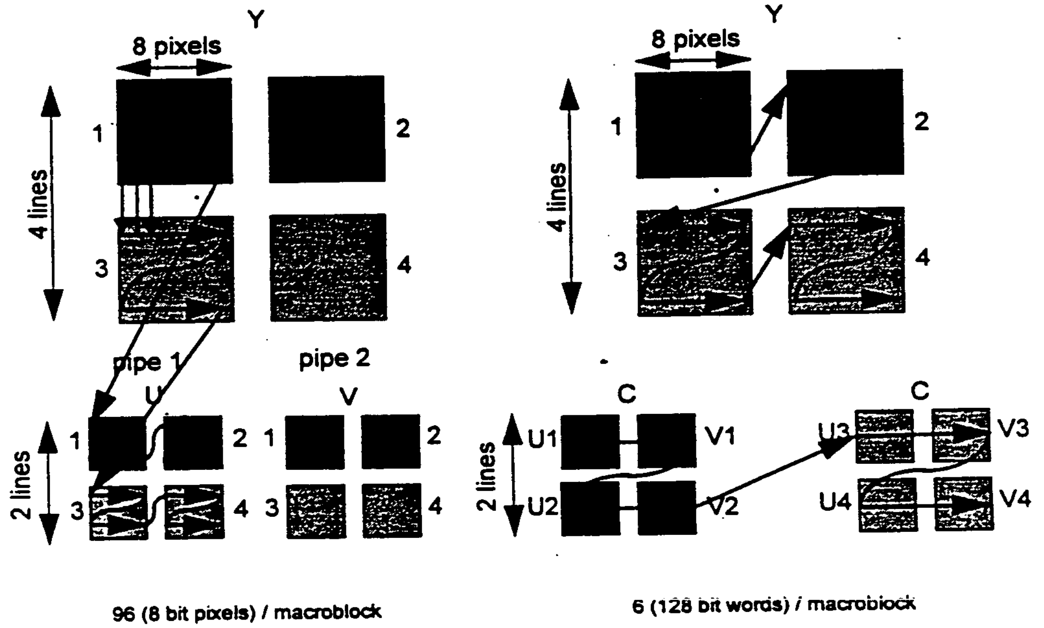
Table 4: Luma write M/2 mode

Word : Contents	Word : Contents	Word : Contents	Word : Contents
1 : Y1	2 : Y2	3 : Y3	4 : Y4

Table 5: Chroma write M/2 mode

Word : Contents	Word : Contents
1 : U1,V1,U2,V2	2 : U3,V3,U4,V4

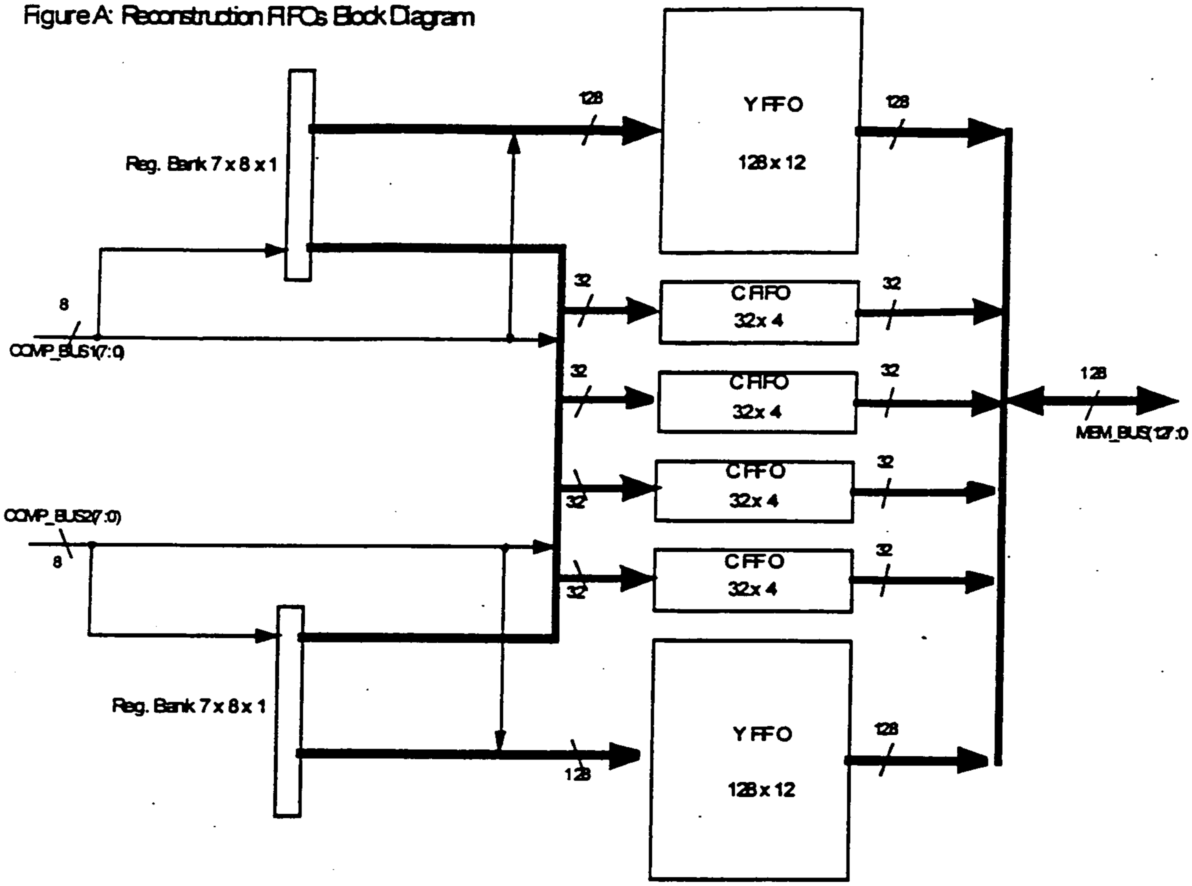
Figure F: Reconstruction fifos M/2 mode (75% compression)



1 MACROBLOCK (Input Pixel Order)

1 MACROBLOCK (Output Pixel Order)

Figure A: Reconstruction FIFOs Block Diagram



APPENDIX W

HD IC GLOBAL SIMULATION PLAN

HD IC Global Simulation Plan

Rev History

vr 1.0 - 2.1	Draft Revisions
vr 2.2	General Distribution
vr 2.3	Corrected typo in 1920x1080 row in section 1 (removed erroneous 59/60fps in DSS mode). Added 720x480 for DSS operation.

HD IC Global Simulation Plan

All Test Items are prioritized from most important [1] to least important [6]

Priortity Levels in Brackets [ ]
1-- Feature required in IC spec and in first product
2-- Feature required in IC spec, but in later product.
3-- Feature required in IC spec, but not currently required in product. Feature may likely be required in future products.
4-- Feature not required in IC spec nor current product, but may be in future product.
5-- Feature required in IC spec and in first product, but functionality is absolutely guaranteed from successful priority 1 operation.
6-- Feature required in IC spec and in later product, but functionality is absolutely guaranteed from successful priority 1, 2, and 3 operations.

1.) Display format conversions

Output Format

Decoded Image Formats SD= 720x480 and Smaller HD= Larger than 720x480	DTC-100 1H NTSC output on 4x3 Display (Displayed Pixels--720x480i at 13.5M pixels/s, 29.97/30Fps)	DTC-100 1H NTSC output on 16x9 Display (Displayed Pixels--720x480i at 13.5M pixels/s, 29.97/30Fps) (Monitor must maintain proper aspect ratio for 4x3 pictures)	DTC100- 2.14H output on 4x3 Display (Displayed Pixels--1920x1080i at 81M pix/s, 29.97/30Fps for HD, 1920x540p at 81M pix/s, 59.94/60Fps for SD)	DTC100- 2.14H output on 16x9 Display (Displayed Pixels--1920x1080i at 81M pix/s, 29.97/30Fps for HD, 1920x540p at 81M pix/s, 59.94/60Fps for SD)	CTC200/201 - 2.14H output on 4x3 Display (Displayed Pixels--1920x1080i at 81M pix/s, 29.97/30Fps for HD, 1920x540p at 81M pix/s, 59.94/60Fps for SD)	CTC200/201 - 2.14H output on 16x9 Display (Displayed Pixels--1920x1080i at 81M pix/s, 29.97/30Fps for HD, 1920x540p at 81M pix/s, 59.94/60Fps for SD)	CTC211 - 2.14H output on 16x9 Display (Displayed Pixels--1920x1080i at 81M pix/s, 29.97/30Fps for HD, 1920x540p at 81M pix/s, 59.94/60Fps for SD)
352x240 (4x3)♠ 23.98/29.97 Fps Int 23.98/29.97 Fps Prog	[1] FS 352x240	[3] SB 544x480 [5] Ana 352x240	[1] FS 352x240	[1] SB 352x240	[5] FS 352x240	[2] SB 352x240 [2] CV [6] Ana 352x240	[6] SB 352x240 [6] CV [6] Ana 352x240
352x240 (16x9)♠ 23.98/29.97 Fps Int 23.98/29.97 Fps Prog	[1] P&S 264x240 [3] LBX	[5] FS 352x240	[1] P&S 264x240	[5] FS 352x240	[5] P&S 264x240 [2] LBX	[6] FS 352x240 [2] CH	[6] FS 352x240 [6] CH
352x480 (4x3)♠ 23.98/29.97 Fps Int 23.98/29.97 Fps Prog	[1] FS 352x240	[3] SB 544x480 [5] Ana 352x480	[1] FS 352x480	[1] SB 352x480	[2] FS 352x480	[2] SB 352x480 [2] CV [6] Ana 352x480	[6] SB 352x480 [6] CV [6] Ana 352x480
352x480 (16x9)♠ 23.98/29.97 Fps Int 23.98/29.97 Fps Prog	[1] P&S 264x240 [3] LBX	[5] FS 352x480	[1] P&S 264x480	[5] FS 352x480	[5] P&S 264x480 [2] LBX	[6] FS 352x480 [2] CH	[6] FS 352x480 [2] CH
480x480 (4x3)♠ 23.98/29.97 Fps Int 23.98/29.97 Fps Prog	[1] FS 480x480	[3] SB 544x480 [5] Ana 480x480	[1] FS 480x480	[1] SB 480x480	[5] FS 480x480	[2] SB 480x480 [2] CV [6] Ana 480x480	[6] SB 480x480 [6] CV [6] Ana 480x480
480x480 (16x9)♠ 23.98/29.97 Fps Int 23.98/29.97 Fps Prog	[1] P&S 360x480 [3] LBX	[5] FS 480x480	[1] P&S 360x480	[5] FS 480x480	[5] P&S 360x480 [2] LBX	[6] FS 480x480 [2] CH	[6] FS 480x480 [5] CH
544x480 (4x3)♠ 23.98/29.97 Fps Int 23.98/29.97 Fps Prog	[1] FS 544x480	[3] SB 544x480 [5] Ana 544x480	[1] FS 544x480	[1] SB 544x480	[5] FS 544x480	[2] SB 544x480 [2] CV [6] Ana 544x480	[6] SB 544x480 [6] CV [6] Ana 544x480
544x480 (16x9)♠ 23.98/29.97 Fps Int 23.98/29.97 Fps Prog	[1] P&S 480x480 [3] LBX	[5] FS 544x480	[1] P&S 480x480	[5] FS 544x480	[5] P&S 480x480 [2] LBX	[6] FS 544x480 [2] CH	[6] FS 544x480 [6] CH
640x480 (4x3)♥ 29.97/30 Fps Int 23.98/24/29.97/30/59.94/60 Prog	[1] FS 640x480	[3] SB 544x480 [5] Ana 640x480	[1] FS 640x480	[1] SB 640x480	[5] FS 640x480	[2] SB 640x480 [2] CV [6] Ana 640x480	[6] SB 640x480 [6] CV [6] Ana 640x480
704x480 (4x3)♠♥	[1] FS 704x480	[3] SB 544x480	[1] FS 704x480	[1] SB 704x480	[5] FS 704x480	[2] SB 704x480	[6] SB 704x480

29.97/30 Fps Int 23.98/24/29.97/30/59.94/60 Prog		[5] Ana 704x480				[2] CV [6] Ana 704x480	[6] CV [6] Ana 704x480
704x480 (16x9)♠♥ 29.97/30 Fps Int 23.98/24/29.97/30/59.94/60 Prog	[1] P&S 544x480 [3] LBX	[1] FS 704x480	[1] P&S 544x480	[5] FS 704x480	[5] P&S 544x480 [2] LBX	[6] FS 704x480 [2] CH	[6] FS 704x480 [6] CH
720x480 (4x3)♠ 29.97/30 Fps Int 23.98/24/29.97/30 Prog	[1] FS 720x480	[3] SB 544x480 [5] Ana 704x480	[1] FS 720x480	[1] SB 720x480	[5] FS 720x480	[2] SB 720x480 [2] CV [6] Ana 720x480	[6] SB 720x480 [6] CV [6] Ana 720x480
720x480 (16x9)♠ 29.97/30 Fps Int 23.98/24/29.97/30 Prog	[1] P&S 544x480 [3] LBX	[1] FS 720x480	[1] P&S 544x480	[5] FS 720x480	[5] P&S 544x480 [2] LBX	[6] FS 720x480 [2] CH	[6] FS 720x480 [6] CH
960x1080 (16x9) 29.97/30 Fps Int 23.98/24/29.97/30 Fps Prog	[4] P&S 720x480 [4] LBX	[4] FS 720x480	[4] P&S 720x1080	[4] FS 960x1080	[4] P&S 720x1080 [4] LBX	[4] FS 960x1080 [4] CH	[4] FS 960x1080 [4] CH
1280x720 (16x9)♥ 23.98/24/29.97/30/59.94/60 Prog	[1] P&S 720x480 [3] LBX	[1] FS 720x480	[1] P&S 960x720♦	[1] FS 1280x720	[2] P&S 960x720♦ [2] LBX	[2] FS 1280x720♦ [2] CH	[6] FS 1280x720 [6] CH
1280x1080 (16x9) 29.97/30 Fps Int 23.98/24/29.97/30 Fps Prog	[4] P&S 720x480 [4] LBX	[4] FS 720x480	[4] P&S 960x1080♦	[4] FS 1280x1080	[4] P&S 960x1080♦ [4] LBX	[4] FS 1280x1080♦ [4] CH	[4] FS 1280x1080 [4] CH
1920x1080 (16x9)♥ 29.97/30 Fps Int 23.98/24/29.97/30 Fps Prog	[1] P&S 720x480 [3] LBX	[1] FS 720x480 [3] SB 544x480	[1] P&S 720x1080	[1] FS 1920x1080	[2] P&S 720x1080 [2] LBX	[2] FS 1920x1080♦ [2] CH♣♦	[6] FS 1920x1080 [6] CH♣
D1 720x480 (4x3) 29.97/30 Fps Int	not applicable	not applicable	not applicable	not applicable	[2] FS 720x480	[2] SB 720x480♦ [2] CV [6] Ana 720x480♦	[6] SB 720x480 [6] CV [6] Ana 720x480
D1 720x480 (16x9) 29.97/30 Fps Int	not applicable	not applicable	not applicable	not applicable	[2] P&S 544x480 [2] LBX	[2] FS 720x480♦ [2] CH	[6] FS 720x480 [6] CH
No Vertical Deflection Zoom HD IC performs both H&V zoom	not applicable	not applicable	not applicable	not applicable	not applicable	[2] Yes ♣	[6] Yes ♣
Vertical Deflection Zoom HD IC performs horizontal zoom Vertical resolution is reducted by deflection/display, not HD IC.	not applicable	not applicable	not applicable	not applicable	not applicable	[2] Yes ♣	[6] Yes ♣

1H NTSC modes are check with Multiplexed YCrCb. 2.14H modes are checked with parallel (non-multiplexed) YCrCb.

♦ Note: Horizontal resolution may be limited to no more than 704 pixels due to filter constraints.

♣ Note: When in ZOOM mode, 1920x1080 format video has an effective resolution of (960/Z) x (1080/Z), where Z=zoom factor

♥ Note: ATSC/GA format

♠ Note: DSS Format

Acronyms:
Ana= Anamorphic video. Picture proportion is not preserved.

SB= Full Vertical Video with "Horizontal Side Bars". Picture proportion is preserved, and all of the decoded image is viewable.
FS= Full Screen. Picture proportion is preserved, all of the decoded image is viewable, and display is filled with active video.
P&S= Pan&Scan. Picture proportion is preserved, some of the decoded image is viewable, and display is filled with active video.
LBX= Letterbox. Picture proportion is preserved, all of the decoded image is viewable.
CH= Zoomed to remove Letterbox, cropped horizontally. H and V resolution is reduced by Zoom Factor from FS res. Picture porportion is preserved.
CV= Zoomed to remove Sidebars, cropped vertically. H and V resolution is reduced by Zoom Factor from FS res. Picture porportion is preserved.
Zoom Factors are determined from the following requirments:

- normal 4x3 image centered on display (default mode)
- normally proportioned 4x3 image to fill 16x9 display (cropped vertical)
- horizontally expanded 4x3 image to fill 16x9 display (no vertical overscan)
- sufficient expansion to allow 1.85x1 formatted video (letterboxed on a 4x3 image) to fill 16x9 display.
- sufficient expansion to allow 1.85x1 formatted video (letterboxed on a 16x9 image) to fill 16x9 display.
- sufficient expansion to allow 2.35x1 formatted video (letterboxed on a 4x3 image) to fill 16x9 display.
- sufficient expansion to allow 2.35x1 formatted video (letterboxed on a 16x9 image) to fill 16x9 display.

General Note1: The video frequency response will begin to roll off before half of the sampling frequency (0.5 Fs) implied by the effective horizontal pixel resolutions given above. The overall digital package frequency response is:

0.0 Fs to 0.3 FS +/- 1 db
0.4 Fs, +2.5 to -4db
0.5 Fs, -18db max

General Note2: The display format conversion chart is meant to be discriptive of IC functionality required in current product definitions. Therefore this chart is not fully decriptive of IC functionality required by the IC specification.

General Note3: 1H NTSC modes are check with Multiplexed YCrCB. 2.14H modes are checked with parallel (non-multiplexed YCrCb).

2.) OSD (checked with transparency and palette variations)

	Display Output Format		
OSD MODE	720x480i, 30fps	1920x1080i, 30fps	1920x540p, 60fps
Compressed Pixel Mode	[1] Hx1, Vx1 [3] Hx2, Hx3, Vx2	[1] Hx2, Vx1 [3] Hx1, Hx3, Vx2	[1] Hx2, Vx1 [3] Hx1, Hx3, Vx2
2-Bit/Pixel Mode	[1] Hx1, Vx1 [3] Hx2, Hx3, Vx2	[1] Hx2, Vx1 [3] Hx1, Hx3, Vx2	[1] Hx2, Vx1 [3] Hx1, Hx3, Vx2
4-Bit/Pixel Mode	[1] Hx1, Vx1 [3] Hx2, Hx3, Vx2	[1] Hx2, Vx1 [3] Hx2, Hx3, Vx2	[1] Hx2, Vx1 [3] Hx2, Hx3, Vx2
True Color Mode without MPEG	[3] Hx1, Hx2, Hx3, Vx1, Vx2	[3] Hx1, Hx2, Hx3, Vx1, Vx2	[3] Hx1, Hx2, Hx3, Vx1, Vx2
True Color Mode with MPEG	[3] Hx1, Hx2, Hx3, Vx1, Vx2	[3] Hx1, Hx2, Hx3, Vx1, Vx2	[3] Hx1, Hx2, Hx3, Vx1, Vx2
Multi Mode, 4-Bit/Pixel with Compressed Pixel Mode	[1] Hx1, Vx1 [3] Hx2, Hx3, Vx2	[1] Hx2, Vx1 [3] Hx1, Hx3, Vx2	[1] Hx2, Vx1 [3] Hx1, Hx3, Vx2
Multi Mode, all four display data types used	[3] Hx1, Hx2, Hx3, Vx1, Vx2	[3] Hx1, Hx2, Hx3, Vx1, Vx2	[3] Hx1, Hx2, Hx3, Vx1, Vx2
Chroma/Luma Override Levels	[1] Yes	[1] Yes	[1] Yes
No OSD nor Video	[1] Yes	[1] Yes	[1] Yes

All display modes checked with worse case (memory bandwidth) MPEG decoding, except where noted.
1H NTSC modes are check with Multiplexed YCrCb. 2.14H modes are checked with parallel (non-multiplexed YCrCb).
All modes are checked with full screen and multiple fractional screen display blocks (no more than one block per line).

3.) Clock

3a.) Clock Switching

[1] Verify switching each Fractional Divider and external input S_CLK0 to internal clock CLK0
[1] Verify switching each Fractional Divider and external input S_CLK1 to internal clock CLK1
[1] Verify switching each Fractional Divider and external input S_CLK2 to internal clock CLK2
[1] Verify switching each Fractional Divider and external input S_CLK3 to internal clock CLK3

[1] Verify switching each Fractional Divider and OFF to external output pin S_CLK0
[1] Verify switching each Fractional Divider and OFF to external output pin S_CLK1
[1] Verify switching each Fractional Divider and OFF to external output pin S_CLK2
[1] Verify switching each Fractional Divider and OFF to external output pin S_CLK3

[1] Verify switching external input S_CLK0 to PLL1
[1] Verify switching external input S_CLK1 to PLL1
[1] Verify switching external input S_CLK2 to PLL1
[1] Verify switching external input S_CLK3 to PLL1

3b.) Clock Generation (S_ClkIn is 27Mhz) Priority [1] items shall be used during all other simulations.

[1] Verify programming CLK0 for 94.5Mhz and 108Mhz
[1] Verify programming CLK1 for 54Mhz
[1] Verify programming CLK2 for 27Mhz, 27.027Mhz, 81Mhz
[1] Verify programming CLK3 for 81Mhz
[3] Verify programming CLK0 for 60Mhz to 120Mhz
[3] Verify programming CLK1 for 45Mhz to 90Mhz
[3] Verify programming CLK2 for 27Mhz to 90Mhz

[3] Verify programming CLK3 for 45Mhz to 120Mhz

4.) Host-Memory Interface

4a.) Memory Read/Write

[1] Walking 1's, walking 0's write to memory (requires read-modify-write)
[1] 1-D Memory Fill with pattern
[1] 2-D Memory Fill with pattern

4b.) Block Copy of pattern
[1] 1-D Block copy
[1] 2-D Block copy, forward
[1] 2-D Block copy, reverse

5.) D1
[2] 720x480 @29.97 and 30Hz interlaced NTSC
[2] Transient sync input. Graceful recovery required.
[4] 720x576 @50Hz interlaced PAL

5.) Raster Generator

5a.) Timing
[1] Must meet timing implied in section 1 and section 2.
[1] Check lines per frame dither for 29.97/30/59.94/60 frames per second.
[4] 720x576 @50Hz interlaced PAL

5b.) Source
[1] Select H and V display sync from external pins on D1 interface. Check display interface.
[1] Select H and V display sync input from external pins on display interface. Check display interface.
[1] Select H and V display sync as output with sync generated internally. Check display interface.

APPENDIX X

EFFECTIVE RESOLUTION IN HxV PIXELS

Effective Resolution in HxV Pixels

Decoded Image Formats	DTC-100 1H NTSC output on 4x3 Display (Displayed Pixels--720x480i at 13.5M pixels/s, 29.97/30Fps)	DTC-100 1H NTSC output on 16x9 Display (Displayed Pixels--720x480i at 13.5M pixels/s, 29.97/30Fps) (Monitor must maintain proper aspect ratio for 4x3 pictures)	DTC100- 2.14H output on 4x3 Display (Displayed Pixels-- 1920x1080i at 81M pixels/s, 29.97/30Fps	DTC100- 2.14H output on 16x9 Display (Displayed Pixels-- 1920x1080i at 81M pixels/s, 29.97/30Fps)	CTC200/201 - 2.14H output on 4x3 Display (Displayed Pixels-- 1920x540p at 81M pixels/s, 59.94/60Fps)	CTC200/201 - 2.14H output on 16x9 Display (Displayed Pixels -- 1920x540p at 81M pixels/s, 59.94/60Fps)	CTC211 - 2.14H output on 16x9 Display (Displayed Pixels -- 1920x1080i at 81M pixels/s, 29.97/30Fps)
352x240 (4x3)♠	352x240	352x240	352x240	352x240	352x240	352x240	352x240
352x240 (16x9)♠	264x240	352x240	264x240	352x240	264x240	352x240	352x240
352x480 (4x3)♠	352x480	352x480	352x480	352x480	352x480	352x480	352x480
352x480 (16x9)♠	264x480	352x480	264x480	352x480	264x480	352x480	352x480
480x480 (4x3)♠	480x480	480x480	480x480	480x480	480x480	480x480	480x480
480x480 (16x9)♠	360x480	480x480	360x480	480x480	360x480	480x480	480x480
544x480 (4x3)♠	544x480	544x480	544x480	544x480	544x480	544x480	544x480
544x480 (16x9)♠	480x480	544x480	480x480	544x480	480x480	544x480	544x480
640x480 (4x3)♥	640x480	640x480	640x480	640x480	640x480	640x480	640x480
704x480 (4x3)♠♥	704x480	704x480	704x480	704x480	704x480	704x480	704x480
704x480 (16x9)♠♥	544x480	704x480	544x480	704x480	544x480	704x480	704x480
960x1080 (16x9, <=30Fps)	720x480	720x480	720x1080	960x1080	720x540	960x540♦	960x1080
1280x720 (16x9)♥	720x480	720x480	960x720♦	1280x720	960x540♦	1280x540♦	1280x720
1280x1080 (16x9, <=30Fps)	720x480	720x480	960x1080♦	1280x1080	960x540♦	1280x540♦	1280x1080
1920x1080 (16x9)♥ (<=30Fps)	720x480	720x480	720x1080	1920x1080	720x540	960x540♦	1920x1080
Deflection Zoom (Z)	not applicable	not applicable	not applicable	not applicable	not applicable	(H/Z)x(V/Z)♦	(H/Z)x(V/Z)♣

♦ Note: Horizontal resolution may be limited to no more than 720 pixels due to filter constraints.

♣ Note: When in ZOOM mode, 1920x1080 format video has an effective resolution of (960/Z) x (1080/Z) in the CTC211.

♥ Note: ATSC/GA format

♠ Note: DSS Format

General Note 2: The video frequency response will begin to roll off before half of the sampling frequency (0.5 Fs) implied by the effective horizontal pixel resolutions given above. The overall digital package frequency response is: 0.0 Fs to 0.3 FS +/- 1 db
0.4 Fs, +2.5 to -4db
0.5 Fs, -18db max

HD Decoder IC memory Bandwidth estimation using NEC uPD 4516161 SDRAM

This sheet: Listing of unrestricted variables

Definitions:

Unrestricted variables: These are variable within specified range. Effects of variations are incorporated in spreadsheet

Restricted variables: Only specified values are allowed. Changing these have global implications, so each case is dealt with in its own spreadsheet

Parameter Block:		**Constants:**	
Unrestricted variables:		frame_pic_type:	0
Mem. Clock rate (MHz):	100	field_pic_type:	1
Mem. CAS Latency (clks):	3	frame_mv:	0
Input Frame Vert. Size:	1088	field_mv:	1
Input Frame Horz. Size:	1920	dual_prime:	2
1 page hit in (accesses):	4	field_16by8:	3
Input Bitrate (bits/s):	83886080	no_comp:	0
OSD Horz. Size:	1920	two_thirds:	1
OSD Vert. Size:	363	mby2_hby2:	2
OSD Mode (2 or 4 bit/pixel):	1		
OSD Resolution:	0		
Percentage of 4 mv MBs:	80	osd_mode_2:	0
Active Video Horz. Size:	1920	osd_mode_4:	1
Active Video Vert. Size:	1088	osd_res_f:	0
Total Video Horz. Size:	2400	osd_res_h:	1
Total Video Vert. Size:	1125	osd_res_t:	2
Display Frame Rate:	30		
On-chip display line stores:	0	yes:	1
SDRAM display line stores:	16	no:	0
Memory page size:	256		
External bus width (bits):	64		

```
'Function macro: Write_clks (   data_size, latency, page_hit, new_page,
'                               hide_head, hide_tail, precharge, transition  )
' Assuming use of precharge to terminate bursts
Function write_clks(data_size, latency, page_hit, new_page, hide_head, hide_tail, precharge, transition)
    page = 256
    over_head = 0
    over_tail = 0
    body = 0

    If hide_head = 0 Then
        'hide_head will be FALSE (=0) for read -> write transitions from
        'the same bank when burst mode is page.  So page mode read->write
        'transition.on same bank will look like non-overlapping accesses.
        'This was made necessary because read and write routines don't
        'include bank switching as part of overhead calculations.  They
        'rely on the hide_head and hide_tail variables to provide this
        'mechanism
        If precharge = 1 Then
            over_head = over_head + 3   'precharge to activation overhead
        End If
        If new_page = 1 Then
            'A new page is deemed to occur everytime the LMU reads from
            'a new area of mem. or if the LMU was forced to terminate
            'the last burst of data using a precharge (in the absence of
            'a pending access request)
            over_head = over_head + 3   'Activation overhead
        End If
    ElseIf transition = 1 Then
        'Following applies to all burst modes and latencies except page
        'mode, where it applies only in cases where the two accesses are
        'from different banks
        over_head = over_head + 1   'read to write Hi-Z cycle overhead
    End If
    If hide_tail = 0 Then
        If latency = 1 Or latency = 2 Then
            over_tail = over_tail + 1 + 3   'last data to precharge to activation
        Else
            over_tail = over_tail + 2 + 3
        End If
    End If
    If page_hit = 1 Then
        'We are ignoring the potential for the LMU to hide this
        'overhead by servicing a request from the other bank
        If latency = 1 Or latency =.2 Then
            body = body + 1 + 3 + 3
        Else
```

```
            body = body + 2 + 3 + 3
        End If
    End If

    write_clks = over_head + over_tail + body + data_size

End Function
```

```
'Function macro: Read_clks (    data_size, latency, page_hit, new_page,
'                               hide_head, hide_tail, precharge, transition  )
' Assuming use of precharge to terminate bursts
Function read_clks(data_size, latency, page_hit, new_page, hide_head, hide_tail, precharge, transition)
    page = 256
    over_head = 0
    over_tail = 0
    body = 0

    If hide_head = 0 Then
        If precharge = 1 Then
            over_head = over_head + 3   'precharge to activation overhead
        End If
        If new_page = 1 Then
            'A new page is deemed to occur everytime the LMU reads from
            'a new area of mem. or if the LMU was forced to terminate
            'the last burst of data using a precharge (in the absence of
            'a pending access request)
            over_head = over_head + 3   'Activation overhead
        End If
        over_head = over_head + latency 'Also applies to write->read xsition
    ElseIf transition = 1 Then
        over_head = over_head + 1       'write to read Hi-Z cycle overhead
    End If
    If hide_tail = 0 Then
        over_tail = over_tail + 3
    End If
    If page_hit = 1 Then
        'We are ignoring the potential for the LMU to hide this
        'overhead by servicing a request from the other bank
        body = body + 3 + 3 + latency
    End If

    read_clks = over_head + over_tail + body + data_size

End Function
```

```
'Function macro: stu_per_mb(    bus_width    )
'
'This function computes the number of storage units in each macroblock
'The bus_widths supported for these calculations are 64 and 128 only.
'bus_width takes values 64 and 128
Function stu_per_mb(b_width)
    Select Case b_width
    Case 64
        stu_per_mb = 4
    Case 128
        stu_per_mb = 2
    End Select

End Function
```

```
'Function macro: block_clks(     comp_mode, bus_width     )
'
'This function computes the number of cycles required to access a storage
'unit from memory.  The definition of a storage unit depends on the
'bus_width parameter.  The bus_widths supported for these calculations are
'64 and 128 only.  For 64 bit bus, the storage unit is one field-block.  For
'128 bit bus, it is two adjacent field blocks.
'comp_mode takes values 0, 1, 2 corresponding to no compression, two-thirds
'and m/2;h/2
'bus_width takes values 64 and 128
Function block_clks(c_mode, b_width)
    Select Case b_width
    Case 64
        luma_pix = 64
        chroma_pix = 16
    Case 128
        If c_mode = 0 Then
            luma_pix = 64
            chroma_pix = 16
        Else
            luma_pix = 128
            chroma_pix = 32
        End If
    End Select

    Select Case c_mode
    Case 0
        'no compression
        block_clks = (luma_pix * 8) / b_width + (chroma_pix * 2 * 8) / b_width
    Case 1
        'two-thirds compression
        block_clks = (luma_pix * 8 * 3) / (4 * b_width) + (chroma_pix * 2 * 8) / (2 * b_width)
    Case 2
        'M/2-H/2 compression
        block_clks = (luma_pix * 8) / (b_width * 4) + (chroma_pix * 2 * 8) / (b_width * 4)
    End Select

End Function
```

```
'Function macro: blk2ras_dec(   stu_clks, latency, stu_per_pg, stu_per_ln, repeat_cnt  )
'
'This function computes the number of cycles required to decode stu_per_ln
'number of field blocks.  These field blocks form the next 8 field lines
'that will be displayed.  The variable stu_per_pg helps calculates the worst
'case number of page hits incurred in reading stu_per_ln field blocks.  The
'variable repeat_cnt is the number of times the basic memory access pattern
'is repeated during a line.  The variable stu_clks is the number of mem. cycles
'required to access one storage unit.  The variable latency is the SDRAM CAS
'latency

Function blk2ras_dec(stu_clks, latency, stu_per_pg, stu_per_ln, repeat_cnt)
    If (Int((stu_per_ln / stu_per_pg))) < (stu_per_ln / stu_per_pg) Then
        num_pg_hits = Int((stu_per_ln / stu_per_pg)) + 1
    Else
        num_pg_hits = stu_per_ln / stu_per_pg
    End If
    If (Int((stu_per_ln / repeat_cnt))) < (stu_per_ln / repeat_cnt) Then
        blks_per_cycle = Int((stu_per_ln / repeat_cnt)) + 1
    Else
        blks_per_cycle = stu_per_ln / repeat_cnt
    End If
    clk_cnt = 0
    blk_cnt = 0
    For cyc_count = 1 To repeat_cnt Step 1
        If (blk_cnt + blks_per_cycle) <= stu_per_ln Then
            If num_pg_hits > 0 Then
                'Very first access in a cycle is new page
                clk_cnt = clk_cnt + read_clks(stu_clks, latency, 0, 1, 0, 0, 1, 0)
                'Assuming blks_per_cycle does not span more than 1 page!
                clk_cnt = clk_cnt + read_clks(stu_clks, latency, 0, 1, 0, 1, 1, 0)
                num_pg_hits = num_pg_hits - 1
                If blks_per_cycle > 2 Then
                    clk_cnt = clk_cnt + read_clks((stu_clks * (blks_per_cycle - 2)), latency, 0, 0, 1, 1, 0, 0)
                End If
            Else
                'Very first access in a cycle is new page
                clk_cnt = clk_cnt + read_clks(stu_clks, latency, 0, 1, 0, 1, 1, 0)
                clk_cnt = clk_cnt + read_clks((stu_clks * (blks_per_cycle - 1)), latency, 0, 0, 1, 1, 0, 0)
            End If
            blk_cnt = blk_cnt + blks_per_cycle
        Else
            If num_pg_hits > 0 Then
                'Very first access in a cycle is new page
                clk_cnt = clk_cnt + read_clks(stu_clks, latency, 0, 1, 0, 0, 1, 0)
                blk_cnt = blk_cnt + 1
                clk_cnt = clk_cnt + read_clks(stu_clks, latency, 0, 1, 0, 1, 1, 0)
                num_pg_hits = num_pg_hits - 1
                blk_cnt = blk_cnt + 1
            Else
                'Very first access in a cycle is new page
                clk_cnt = clk_cnt + read_clks(stu_clks, latency, 0, 1, 0, 1, 1, 0)
                blk_cnt = blk_cnt + 1
            End If
            If blk_cnt < stu_per_ln Then
                clk_cnt = clk_cnt + read_clks((stu_clks * (stu_per_ln - blk_cnt)), latency, 0, 0, 1, 1, 0, 0)
                blk_cnt = stu_per_ln
            End If
        End If
    Next cyc_count
    blk2ras_dec = clk_cnt
End Function
```

```
'Function macro: osd_clks(  b_width, h_size, v_size, mode, res, frame_lines )
'
'This function computes the number of memory cycles required per
'scan line to satisfy reading or writing of OSD information from
'or to the SDRAM.  Variable b_width denotes the SDRAM bus bandwidth
'Variables h_size and v_size represent the worst case horizontal and
'vertical sizes of OSD.  Variable mode refers to 2 bit/pixel (=0) or
'4 bit/pixel (1) OSD.  Variable res refers to OSD resolution: 0=>full
';1=>half and 2=>one-third.  Variable frame_lines refers to the number
'of lines between two odd synchs

Function osd_clks(b_width, h_size, v_size, mode, res, frame_lines)
    pel_area = h_size * v_size
    Select Case b_width
    Case 64
        Select Case mode
        Case 0
            Select Case res
            Case 0
                    frame_clks = pel_area / 32
            Case 1
                    frame_clks = pel_area / 64
            Case 2
                    frame_clks = pel_area / 96
            End Select
        Case 1
            Select Case res
            Case 0
                    frame_clks = pel_area / 16
            Case 1
                    frame_clks = pel_area / 32
            Case 2
                    frame_clks = pel_area / 48
            End Select
        End Select
    Case 128
        Select Case mode
        Case 0
            Select Case res
            Case 0
                    frame_clks = pel_area / 64
            Case 1
                    frame_clks = pel_area / 128
            Case 2
                    frame_clks = pel_area / 192
            End Select
        Case 1
            Select Case res
            Case 0
                    frame_clks = pel_area / 32
            Case 1
                    frame_clks = pel_area / 64
            Case 2
                    frame_clks = pel_area / 96
            End Select
        End Select
    End Select
    If (Int(frame_clks / frame_lines)) < (frame_clks / frame_lines) Then
        osd_clks = Int(frame_clks / frame_lines) + 1
    Else
        osd_clks = frame_clks / frame_lines
    End If
End Function
```

```
'Function macro: raster_rd_wr(  act_ln_pels, bus_width, stu_per_ln, latency )
'
'This function computes the number of cycles required to perform paired
'read-write operations of 4:2:0 format display data.  Each line, a raster
'line of display luma and half a raster line of display chroma are read
'out of one bank, while the same number of luma and chroma pels are written
'into the opposite bank from the display section decode of stu_per_ln
'storage units.  The variable act_ln_pels represents the number of active luma
'pels in each line.  Variables bus_width and latency denote SDRAM bus width and
'CAS latency.

Function raster_rd_wr(act_ln_pels, bus_width, stu_per_ln, latency)

    clk_cnt = 0
    write_cnt = 0
    read_cnt = 0
    rd_pel_cnt = 0
    wr_pel_cnt = 0
    Select Case bus_width
    Case 64
        'Each storage unit is a field block - an 8 by 8 luma block with 2 4 by 4
        'chroma blocks
        'No matter what the latency, PRE-ACTIVE (3 clks) and ACTIVE-READ/WRITE
        '(3 clks) will mandate 8 access cycles for zero overhead bank mux'ed
        'operations.  The paired bank mux'ed operations should escape from first
        'access and last access overhead if the LMC algorithm is synchronous enough
        'Active line pels is always a multiple of 16
        While ((write_cnt + 8) <= stu_per_ln) And ((read_cnt + 8) <= (Int(act_ln_pels / 64)))
            'Assume worst case, that every pair induces a read<->write transition
            For luma_blk = 1 To 8 Step 1
                clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 1, 1, 1, 1)
                clk_cnt = clk_cnt + write_clks(8, latency, 0, 1, 1, 1, 1, 1)
            Next luma_blk
            For chroma_blk = 1 To 4 Step 1
                clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 1, 1, 1, 1)
                clk_cnt = clk_cnt + write_clks(8, latency, 0, 1, 1, 1, 1, 1)
            Next chroma_blk
            write_cnt = write_cnt + 8
            read_cnt = read_cnt + 8
            rd_pel_cnt = rd_pel_cnt + 512
            wr_pel_cnt = wr_pel_cnt + 512
        Wend
        While ((rd_pel_cnt + 128) <= act_ln_pels) And ((wr_pel_cnt + ((stu_per_ln - write_cnt) * 16)) <= (stu_per_
ln * 64))
            'Two luma accesses and ...
            clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 1, 1, 1, 1)
            clk_cnt = clk_cnt + write_clks((stu_per_ln - write_cnt), latency, 0, 1, 1, 0, 1, 1)
            clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 1, 1, 1, 1)
            clk_cnt = clk_cnt + write_clks((stu_per_ln - write_cnt), latency, 0, 1, 1, 0, 1, 1)
            '... one chroma access
            clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 1, 1, 1, 1)
            clk_cnt = clk_cnt + write_clks((stu_per_ln - write_cnt), latency, 0, 1, 1, 0, 1, 1)
            rd_pel_cnt = rd_pel_cnt + 128
            wr_pel_cnt = wr_pel_cnt + (stu_per_ln - write_cnt) * 16
        Wend
        If rd_pel_cnt < act_ln_pels Then
            clk_cnt = clk_cnt + read_clks(((act_ln_pels - rd_pel_cnt) / 8), latency, 0, 1, 0, 1, 1, 1)
            clk_cnt = clk_cnt + read_clks(((act_ln_pels - rd_pel_cnt) / 16), latency, 0, 1, 0, 1, 1, 1)
        End If
        While wr_pel_cnt < (stu_per_ln * 64)
            'Two luma field block lines and ...
            clk_cnt = clk_cnt + write_clks(((stu_per_ln - write_cnt) * 2), latency, 0, 1, 0, 1, 1, 1)
            '...one chroma field block line
            clk_cnt = clk_cnt + write_clks((stu_per_ln - write_cnt), latency, 0, 1, 0, 1, 1, 1)
            wr_pel_cnt = wr_pel_cnt + (stu_per_ln - write_cnt) * 16
        Wend

    Case 128
        'Each storage unit is two horizontally adjascent field blocks
        'No matter what the latency, PRE-ACTIVE (3 clks) and ACTIVE-READ/WRITE
        '(3 clks) will mandate 8 access cycles for zero overhead bank mux'ed
        'operations.  The paired bank mux'ed operations should escape from first
        'access and last access overhead if the LMC algorithm is synchronous enough
        'Active line pels is always a multiple of 16
        While ((write_cnt + 8) <= stu_per_ln) And ((read_cnt + 8) <= (Int(act_ln_pels / 128)))
            'Assume worst case, that every pair induces a read<->write transition
            For luma_blk = 1 To 8 Step 1
                clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 1, 1, 1, 1)
                clk_cnt = clk_cnt + write_clks(8, latency, 0, 1, 1, 1, 1, 1)
            Next luma_blk
            For chroma_blk = 1 To 4 Step 1
                clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 1, 1, 1, 1)
                clk_cnt = clk_cnt + write_clks(8, latency, 0, 1, 1, 1, 1, 1)
            Next chroma_blk
            write_cnt = write_cnt + 8
            read_cnt = read_cnt + 8
            rd_pel_cnt = rd_pel_cnt + 1024
            wr_pel_cnt = wr_pel_cnt + 1024
        Wend
        While ((rd_pel_cnt + 256) <= act_ln_pels) And ((wr_pel_cnt + ((stu_per_ln - write_cnt) * 32)) <= (stu_per_
ln * 128))
            'Two luma accesses and ...
            clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 1, 1, 1, 1)
            clk_cnt = clk_cnt + write_clks((stu_per_ln - write_cnt), latency, 0, 1, 1, 0, 1, 1)
            clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 1, 1, 1, 1)
            clk_cnt = clk_cnt + write_clks((stu_per_ln - write_cnt), latency, 0, 1, 1, 0, 1, 1)
            '... one chroma access
            clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 1, 1, 1, 1)
            clk_cnt = clk_cnt + write_clks((stu_per_ln - write_cnt), latency, 0, 1, 1, 0, 1, 1)
            rd_pel_cnt = rd_pel_cnt + 256
            wr_pel_cnt = wr_pel_cnt + (stu_per_ln - write_cnt) * 32
        Wend
        If rd_pel_cnt < act_ln_pels Then
            clk_cnt = clk_cnt + read_clks(((act_ln_pels - rd_pel_cnt) / 16), latency, 0, 1, 0, 1, 1, 1)
            clk_cnt = clk_cnt + read_clks(((act_ln_pels - rd_pel_cnt) / 32), latency, 0, 1, 0, 1, 1, 1)
        End If
        While wr_pel_cnt < (stu_per_ln * 128)
            'Two luma field block lines and ...
            clk_cnt = clk_cnt + write_clks(((stu_per_ln - write_cnt) * 2), latency, 0, 1, 0, 1, 1, 1)
            '...one chroma field block line
            clk_cnt = clk_cnt + write_clks((stu_per_ln - write_cnt), latency, 0, 1, 0, 1, 1, 1)
            wr_pel_cnt = wr_pel_cnt + (stu_per_ln - write_cnt) * 32
        Wend
    End Select
    raster_rd_wr = clk_cnt
End Function
```

```
'Function macro: vld_clks(  b_width, bitrate, frame_rate, frame_lines    )
'
'This function computes the number of memory cycles required per
'scan line to satisfy reading or writing of compressed bitstream
'from or to the bit buffer

Function vld_clks(b_width, bitrate, frame_rate, frame_lines)

    If (Int(bitrate / (b_width * frame_rate * frame_lines))) < (bitrate / (b_width * frame_rate * frame_lines)) Th
en
        vld_clks = Int(bitrate / (b_width * frame_rate * frame_lines)) + 1
    Else
        vld_clks = bitrate / (b_width * frame_rate * frame_lines)
    End If

End Function
```

```
'Function macro: osd_vld_rd_wr(  osd_per_ln, vld_per_ln, latency    )
'
'This function computes the number of cycles required to perform paired
'read-write operations of OSD and VLD data.  Each line, a portion of data from
'a bitstream input FIFO is transferred to SDRAM.  At the same time, an OSD
'input FIFO is also serviced if data is available.  The two transfers occur in
'opposite memory banks.  So overhead can be hidden, provided each session is 8
'bus cycles long.  Variables osd_per_ln and vld_per_ln denote the number of OSD
'and VLD read and write cycles that are needed per scan line.  Variable latency
'denotes SDRAM CAS latency.

Function osd_vld_rd_wr(osd_per_ln, vld_per_ln, latency)

    clk_cnt = 0
    osd_rd_cnt = 0
    osd_wr_cnt = 0
    vld_rd_cnt = 0
    vld_wr_cnt = 0

    'No matter what the latency, PRE-ACTIVE (3 clks) and ACTIVE-READ/WRITE
    '(3 clks) will mandate 8 access cycles for zero overhead bank mux'ed
    'operations.  The paired bank mux'ed operations should escape from first
    'access and last access overhead if the LMC algorithm is synchronous enough
    'First access to account for page hit overhead
    If (osd_per_ln > 8) Then
        clk_cnt = clk_cnt + read_clks(8, latency, 1, 0, 1, 1, 1, 1)
        clk_cnt = clk_cnt + write_clks(8, latency, 1, 0, 1, 1, 1, 1)
        osd_rd_cnt = osd_rd_cnt + 8
        osd_wr_cnt = osd_wr_cnt + 8
    Else
        clk_cnt = clk_cnt + read_clks(osd_per_ln, latency, 1, 0, 0, 1, 1, 1)
        clk_cnt = clk_cnt + write_clks(osd_per_ln, latency, 1, 0, 0, 1, 1, 1)
        osd_rd_cnt = osd_per_ln
        osd_wr_cnt = osd_per_ln
    End If
    If (vld_per_ln > 8) Then
        clk_cnt = clk_cnt + read_clks(8, latency, 1, 0, 1, 1, 1, 0)
        clk_cnt = clk_cnt + write_clks(8, latency, 1, 0, 1, 1, 1, 0)
        vld_rd_cnt = vld_rd_cnt + 8
        vld_wr_cnt = vld_wr_cnt + 8
    Else
        clk_cnt = clk_cnt + read_clks(vld_per_ln, latency, 1, 0, 0, 1, 1, 0)
        clk_cnt = clk_cnt + write_clks(vld_per_ln, latency, 1, 0, 0, 1, 1, 0)
        vld_rd_cnt = vld_per_ln
        vld_wr_cnt = vld_per_ln
    End If
```

```
While ((osd_rd_cnt + 8) <= osd_per_ln) And ((vld_rd_cnt + 8) <= vld_per_ln)
    'Worst case both reads are inserted between two writes.  But we assume
    'more intelligence on the part of the LMC.   Note that this worst case
    'situation cannot be represented in this model as the trailing write has
    'no idea about these read operations
    clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 1, 1, 1, 1)
    clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 1, 1, 1, 0)
    osd_rd_cnt = osd_rd_cnt + 8
    vld_rd_cnt = vld_rd_cnt + 8
Wend
While ((osd_wr_cnt + 8) <= osd_per_ln) And ((vld_wr_cnt + 8) <= vld_per_ln)
    'See comments above
    clk_cnt = clk_cnt + write_clks(8, latency, 0, 1, 1, 1, 1, 1)
    clk_cnt = clk_cnt + write_clks(8, latency, 0, 1, 1, 1, 1, 0)
    osd_wr_cnt = osd_wr_cnt + 8
    vld_wr_cnt = vld_wr_cnt + 8
Wend
While ((osd_wr_cnt + 8) <= osd_per_ln) And ((vld_rd_cnt + 8) <= vld_per_ln)
    clk_cnt = clk_cnt + write_clks(8, latency, 0, 1, 1, 1, 1, 1)
    clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 1, 1, 1, 1)
    osd_wr_cnt = osd_wr_cnt + 8
    vld_rd_cnt = vld_rd_cnt + 8
Wend
While ((osd_rd_cnt + 8) <= osd_per_ln) And ((vld_wr_cnt + 8) <= vld_per_ln)
    clk_cnt = clk_cnt + write_clks(8, latency, 0, 1, 1, 1, 1, 1)
    clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 1, 1, 1, 1)
    osd_rd_cnt = osd_rd_cnt + 8
    vld_wr_cnt = vld_wr_cnt + 8
Wend
While (osd_rd_cnt + 8) <= osd_per_ln
    clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 0, 1, 1, 0)
    osd_rd_cnt = osd_rd_cnt + 8
Wend
While (osd_wr_cnt + 8) <= osd_per_ln
    clk_cnt = clk_cnt + write_clks(8, latency, 0, 1, 0, 1, 1, 0)
    osd_wr_cnt = osd_wr_cnt + 8
Wend
While (vld_rd_cnt + 8) <= vld_per_ln
    clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 0, 1, 1, 0)
    vld_rd_cnt = vld_rd_cnt + 8
Wend
While (vld_wr_cnt + 8) <= vld_per_ln
    clk_cnt = clk_cnt + write_clks(8, latency, 0, 1, 0, 1, 1, 0)
    vld_wr_cnt = vld_wr_cnt + 8
Wend
If osd_rd_cnt < osd_per_ln Then
```

```
        clk_cnt = clk_cnt + read_clks((osd_per_ln - osd_rd_cnt), latency, 0, 1, 0, 1, 1, 0)
    End If
    If osd_wr_cnt < osd_per_ln Then
        clk_cnt = clk_cnt + write_clks((osd_per_ln - osd_wr_cnt), latency, 0, 1, 0, 1, 1, 0)
    End If
    If vld_rd_cnt < vld_per_ln Then
        clk_cnt = clk_cnt + read_clks((vld_per_ln - vld_rd_cnt), latency, 0, 1, 0, 1, 1, 0)
    End If
    If vld_wr_cnt < vld_per_ln Then
        clk_cnt = clk_cnt + write_clks((vld_per_ln - vld_wr_cnt), latency, 0, 1, 0, 1, 1, 0)
    End If

    osd_vld_rd_wr = clk_cnt

End Function
```

```
'Function macro: pred_wc_clks(    b_width, latency, stu_clks, mbs_per_ln  )
'
'This function computes the number of clock cycles required per H reset
'to read in 4 (worst case) predictors from memory, and write each macroblock
'computed in that interval.  All predictors are further assumed to require
'data from the same bank, so overhead savings are minimal.  Variables b_width
'and latency denote the bus width and CAS latency of the target SDRAM.
'Variable stu_clks represents the number of memory cycles required to access
'one storage unit in memory.  Variable mbs_per_ln represents the number of
'macroblocks processed per H reset

Function pred_wc_clks(b_width, latency, stu_clks, mbs_per_ln)

    clk_cnt = 0

    Select Case b_width
    Case 64
        'A storage unit is one 8 by 8 field block.  Six such st. units are
        'accessed to form one predictor.  Each predictor, given the memory
        'organization, can have at most one page crossing (in the upper 3
        'or lower 3 blocks but not both at the same time)

        'First macroblock in scan line ...
        'First field upper half ...
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 1, 1, 0, 0, 1, 1)
        '... Second field upper half ...
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 1, 1, 0, 0, 1, 0)
        '... First field lower half ...
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 0, 0, 1, 0)
        '... Second field lower half ...
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 0, 0, 1, 0)
        '... and similarly for the other two predictors:
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 1, 1, 0, 0, 1, 0)
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 1, 1, 0, 0, 1, 0)
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 0, 0, 1, 0)
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 0, 1, 1, 0)
        'Writing 4 storage units to memory:
        'First writes in each field (writes are too small to hide overhead) ...
        clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 1, 1, 1, 1)
        clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 0, 1, 1, 0)
        'Second writes in each field with page crossing (need precharge)
        clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 0, 1, 1, 0)
        clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 0, 1, 1, 0)

        'Intermediete macroblocks in scan line ....
        For i = 1 To (mbs_per_ln - 2) Step 1
        'First field upper half ...
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 1, 1, 0, 0, 1, 1)
        '... Second field upper half ...
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 1, 1, 0, 0, 1, 0)
        '... First field lower half ...
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 0, 0, 1, 0)
        '... Second field lower half ...
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 0, 0, 1, 0)
        '... and similarly for the other two predictors:
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 1, 1, 0, 0, 1, 0)
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 1, 1, 0, 0, 1, 0)
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 0, 0, 1, 0)
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 0, 1, 1, 0)
        'Writing 4 storage units to memory:
        'First writes in each field ...
        clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 1, 1, 1, 1)
        clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 0, 1, 1, 0)
        'Second writes in each field without page crossing (no precharge)
        clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 0, 0, 1, 0, 0)
        clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 0, 0, 1, 0, 0)
    Next i
    'Last macroblock in scan line
    'First field upper half ...
    clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 1, 1, 0, 0, 1, 1)
    '... Second field upper half ...
    clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 1, 1, 0, 0, 1, 0)
    '... First field lower half ...
    clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 0, 0, 1, 0)
    '... Second field lower half ...
    clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 0, 0, 1, 0)
    '... and similarly for the other two predictors:
    clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 1, 1, 0, 0, 1, 0)
    clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 1, 1, 0, 0, 1, 0)
    clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 0, 0, 1, 0)
    clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 0, 1, 1, 0)
    'Writing 4 storage units to memory:
    'First writes in each field ...
    clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 1, 1, 1, 1)
    clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 0, 1, 1, 0)
    'Second writes in each field without page crossing (no precharge)
    clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 0, 0, 1, 0, 0)
    clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 0, 0, 0, 0, 0)
Case 128
    'A storage unit is 2 adjascent 8 by 8 field blocks.  Four such st. units
    'are accessed to form one predictor.  Each predictor, given the mem.
    'organization, can have at most one page crossing (in the upper 2 units
'or the lower 2 units but not both at the same time)
'Number of read bus cycles are not large enough to fully hide overhead
'First macroblock in scan line ...
'First field upper half ...
clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 0, 0, 1, 1)
'... Second field upper half ...
clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 0, 0, 1, 0)
'... First field lower half ...
clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 0, 0, 1, 0)
'... Second field lower half ...
clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 0, 0, 1, 0)
'... and similarly for the other two predictors:
clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 0, 0, 1, 0)
clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 0, 0, 1, 0)
clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 0, 0, 1, 0)
clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 0, 1, 1, 0)
'Writing 2 storage units to memory:
clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 0, 1, 1, 1)
clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 0, 1, 1, 0)

'Intermediete macroblock in scan line ...
For i = 1 To (mbs_per_ln - 2) Step 1
    'First field upper half ...
    clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 0, 0, 1, 1)
    '... Second field upper half ...
    clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 0, 0, 1, 0)
    '... First field lower half ...
    clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 0, 0, 1, 0)
    '... Second field lower half ...
    clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 0, 0, 1, 0)
    '... and similarly for the other two predictors:
    clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 0, 0, 1, 0)
    clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 0, 0, 1, 0)
    clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 0, 0, 1, 0)
    clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 0, 1, 1, 0)
    'Writing 2 storage units to memory:
    clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 0, 1, 1, 1)
    clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 0, 1, 1, 0)
Next i

'And the last macroblock
'First field upper half ...
clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 0, 0, 1, 1)
'... Second field upper half ...
clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 0, 0, 1, 0)
'... First field lower half ...
```

```
        '... Second field upper half ...
        clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 1, 0, 1, 0)
        '... First field lower half ...
        clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 1, 0, 1, 0)
        '... Second field lower half ...
        clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 1, 0, 1, 0)
        '... and similarly for the other two predictors:
        clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 1, 0, 1, 0)
        clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 1, 0, 1, 0)
        clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 1, 0, 1, 0)
        clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 1, 0, 1, 0)
        'Writing 2 storage units to memory:
        clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 1, 1, 1, 1)
        clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 0, 0, 1, 0)
    End Select

    pred_wc_clks = clk_cnt

End Function
```

```
'Function macro: pred_bc_clks(    b_width, latency, stu_clks, mbs_per_ln  )
'
'This function computes the number of clock cycles required per H reset
'to read in 2 (best case) predictors from memory (forward only frame pred-
'ictor), and write each macroblock computed in that interval.  Each predictor
'is, by design, accessed from opposite banks of SDRAM.  Variables b_width
'and latency denote the bus width and CAS latency of the target SDRAM.
'Variable stu_clks represents the number of memory cycles required to access
'one storage unit in memory.  Variable mbs_per_ln represents the number of
'macroblocks processed per H reset

Function pred_bc_clks(b_width, latency, stu_clks, mbs_per_ln)

    clk_cnt = 0

    Select Case b_width
    Case 64
        'A storage unit is one 8 by 8 field block.  Six such st. units are
        'accessed to form one predictor.  Each predictor, given the memory
        'organization, can have at most one page crossing (in the upper 3
        'or lower 3 blocks but not both at the same time)

        'First macroblock in scan line ...
        'First field upper half ...
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 1, 1, 0, 1, 1, 1).
        '... Second field upper half ...
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 1, 1, 1, 1, 1, 0)
        '... First field lower half ...
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 1, 1, 1, 0)
        '... Second field lower half ...
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 1, 1, 1, 0)

        'Writing 4 storage units to memory:
        'Writes are not large enough to hide overhead
        'First writes in each field ...
        clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 1, 1, 1, 1)
        clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 0, 1, 1, 0)
        'Second writes in each field with page crossing (need precharge)
        clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 0, 1, 1, 0)
        clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 0, 1, 1, 0)

        'Intermediete macroblocks in scan line ....
        For i = 1 To (mbs_per_ln - 2) Step 1
            'First field upper half ...
            clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 1, 1, 0, 1, 1, 1)
            '... Second field upper half ...
            clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 1, 1, 1, 1, 1, 0)
            '... First field lower half ...
            clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 1, 1, 1, 0).
            '... Second field lower half ...
            clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 1, 1, 1, 0)

            'Writing 4 storage units to memory:
            'First writes in each field ...
            clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 1, 1, 1, 1)
            clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 0, 1, 1, 0)
            'Second writes in each field without page crossing (no precharge)
            clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 0, 0, 1, 0, 0)
            clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 0, 0, 1, 0, 0)
        Next i
        'Last macroblock in scan line
        'First field upper half ...
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 1, 1, 0, 1, 1, 1)
        '... Second field upper half ...
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 1, 1, 1, 1, 1, 0)
    '... First field lower half ...
    clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 1, 1, 1, 0)
    '... Second field lower half ...
    clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 1, 1, 1, 0)

    'Writing 4 storage units to memory:
    'First writes in each field ...
    clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 1, 1, 1, 1)
    clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 0, 1, 1, 0)
    'Second writes in each field without page crossing (no precharge)
    clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 0, 0, 1, 0, 0)
    clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 0, 0, 0, 0, 0)
Case 128
    'A storage unit is 2 adjascent 8 by 8 field blocks.  Four such st. units
    'are accessed to form one predictor.  Each predictor, given the mem.
    'organization, can have at most one page crossing (in the upper 2 units
    'or the lower 2 units but not both at the same time)
    'Number of read bus cycles are not large enough to fully hide overhead
    'Note:  Tail overhead gives a more meaningful number of overhead cycles
    '       This is not representative of what is assumed to be happening
    'First macroblock in scan line ...
    'First field upper half ...
    clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 0, 1, 1, 1)
    '... Second field upper half ...
    clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 1, 0, 1, 0)
    '... First field lower half ...
    clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 1, 0, 1, 0)
    '... Second field lower half ...
    clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 1, 0, 1, 0)

    'Writing 2 storage units to memory:
    'Writes are not large enough to hide overhead
    clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 1, 1, 1, 1)
    clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 0, 1, 1, 0)

    'Intermediete macroblock in scan line ...
    For i = 1 To (mbs_per_ln - 2) Step 1
        'First field upper half ...
        clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 0, 1, 1, 1)
        '... Second field upper half ...
        clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 1, 0, 1, 0)
        '... First field lower half ...
        clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 1, 0, 1, 0)
        '... Second field lower half ...
        clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 1, 0, 1, 0)

        'Writing 2 storage units to memory:
        clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 1, 1, 1, 1)
        clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 0, 1, 1, 0)
    Next i

    'And the last macroblock
    'First field upper half ...
    clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 0, 1, 1, 1)
    '... Second field upper half ...
    clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 1, 0, 1, 0)
    '... First field lower half ...
    clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 1, 0, 1, 0)
    '... Second field lower half ...
    clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 1, 0, 1, 0)

    'Writing 2 storage units to memory:
    clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 1, 1, 1, 1)
    clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 0, 0, 1, 0)
End Select
```

```
    pred_bc_clks = clk_cnt
End Function
```

HD Decoder IC memory Bandwidth estimation using NEC uPD 4516161 SDRAM

This sheet: Computation of memory bandwidth loads for one set of restricted variables

Assumptions:

1) Each reconstructed frame is stored as two independent fields in memory

2) Luma-chroma data from one field in a macroblock is saved as a storage unit

3) Reconstructed storage units in memory do not cross page boundaries

4)Each reconstructed field memory area starts at a page boundary

Restricted variables:		Worst case Scenario:	
		1) All MBs in a line are coded with dual prime motion vectors (4 predictors)	
		2) OSD information is being updated with page hit	
Picture type (0=>frame):	0	3) OSD information is being read for display with page hit	
Motion Estimation mode:	2	4) One word of PES header data is read out	
Compression mode:	2	5) Every set of predictors crosses a page boundary	
		6) Compressed bitstream is being read out with page hit	
		7) Compressed bitstream is being written with page hit	
Tot. MBs in input frame:	8160	OSD read or update cycles per H reset:	39
Frame time (ns):	33333333	Bit buffer read or write cycles per H reset:	39
MBs processed per H_reset (High BW):	8	OSD & VLD read-write cycles per H reset:	228
MBs processed per H_reset (Low BW):	7	PES Header read cycles per H reset:	20
Storage Unit access clocks:	3	Refresh overhead per field:	2048
Storage Units per macroblock:	4		
Storage Units per page:	85	Sum total read & write cycles per H reset (High BW/Wrst Case):	2338
		Sum total read & write cycles per H reset (Low BW/Wrst Case):	2204
		Sum total read & write cycles per H reset (High BW/Best Case):	1962
		Sum total read & write cycles per H reset (Low BW/Best Case):	1874
MB predictor read cycles per H reset (High BW/Wrst Case):	1049	Per field total bandwidth including refresh:	1259630
MB predictor read cycles per H reset (Low BW/Wrst Case):	921	Memory bus clock frequency:	75577800
Storage Units for display per H reset:	30		
Display decoder read cycles per H reset (High BW):	174	Bandwidth calculations assuming on-chip block-to-scan conversion:	
Display decoder read cycles per H reset (Low BW):	168	Sum total read & write cycles per H reset (High BW/Wrst Case):	1471
Display line buffer read and write:	867	Sum total read & write cycles per H reset (Low BW/Wrst Case):	1337
MB predictor read cycles per H reset (High BW/Best Case):	673	Sum total read & write cycles per H reset (High BW/Best Case):	1095
MB predictor read cycles per H reset (Low BW/Best Case):	591	Sum total read & write cycles per H reset (Low BW/Best Case):	1007
		Per field total bandwidth including refresh:	771942
		Memory bus clock frequency:	46316520

APPENDIX Y

MODES.XLS - NON-HD AND HD

	Input Format										
			Frame	External	Memory	Free	Compression	Mem Clk Rate	Display requires		
Display Type	Hor	Vert	Rate	Memory	Required	Memory	Mode	Required	lines:	LMU	Memory Map
480 progressive	720	480	D1 input	32 Mbits	1.4	30.6	-			Yes	
" "	352	240	?	"	13.9	18.1	-			-	
" "	352	480	30 fps	"	17.0	15.0	-			Yes	
" "	720	480	"	"	23.3	8.7	-			Yes	
" "	352	480	60 fps	"	17.0	15.0	-		frame	-	
" "	720	480	"	"	23.3	8.7	-			-	
" "	1280	720	60 fps	"	19.2	12.8	H/2 M/2		frame	-	
" "	1920	1088	30 fps	"	29.8	2.2	H/2 M/2		field	-	
480 interlaced	720	480	D1 Input	32 Mbits	1.4	30.6	-			-	
" "	352	240	?	"	23.3	8.7	-			-	
" "	352	480	30 fps	"	17.0	15.0	-			-	
" "	720	480	30 fps	"	23.3	8.7	-			-	
" "	352	480	60 fps	"	17.0	15.0	-		frame	-	
" "	720	480	60 fps	"	23.3	8.7	-			-	
" "	1280	720	60 fps	"	19.2	12.8	H/2 M/2	100.2	frame	-	
" "	1920	1088	30 fps	"	29.8	2.2	H/2 M/2	96.5	field	-	

Mbit= 10^6 bits

manufacture or sale of apparatus or devices without permission.

Display Type	Input Format			External	**		***	OSD	LMU	Total	Free
			Frame				Display				
	Hor	Vert	Rate	Memory	3 Frames	Bit Buffer	Buffer	Mem.	Mem.	Memory	Memory
480 progressive	720	480	D1 input	32	-	-	-	1.4	5.5	1.4	30.6
" "	352	240	?	32	3.0	9.5		1.4	-	13.9	18.1
" "	352	480	30 fps	32	6.1	9.5		1.4	2.7	17.0	15.0
" "	720	480	"	32	12.4	9.5		1.4	5.5	23.3	8.7
" "	352	480	60 fps	32	6.1	9.5		1.4	-	17.0	15.0
" "	720	480	"	32	12.4	9.5		1.4	-	23.3	8.7
" "	1280	720	60 fps	32	8.3	9.5	0.1	1.4	-	19.2	12.8
" "	1920	1088	30 fps	32	18.8	9.5	0.09	1.4	-	29.8	2.2
480 interlaced	720	480	D1 Input	32	-	-		1.4	-	1.4	30.6
" "	352	240	?	32	12.4	9.5		1.4	-	23.3	8.7
" "	352	480	30 fps	32	6.1	9.5		1.4	-	17.0	15.0
" "	720	480	30 fps	32	12.4	9.5		1.4	-	23.3	8.7
" "	352	480	60 fps	32	6.1	9.5		1.4	-	17.0	15.0
" "	720	480	60 fps	32	12.4	9.5		1.4	-	23.3	8.7
" "	1280	720	60 fps	32	8.3	9.5	0.1	1.4	-	19.2	12.8
" "	1920	1088	30 fps	32	18.8	9.5	0.09	1.4	-	29.8	2.2

Mbit= 10^6 bits

** Include compression

*** In H/2 Mode, uses half H res

THOMSON CONSUMER ELECTRONICS
CONFIDENTIAL AND PROPRIETARY
These drawings and specifications are the property of Thomson Consumer Electronics Inc. and shall not be reproduced or copied or used as the basis for manufacture or sale of apparatus or devices without

Greg K

manufacture or sale of apparatus or devices without permission.

	Input Format							Mem Clk	Display		
			Frame	External	Memory	Free	Compression	Rate	requires		
Display Type	Hor	Vert	Rate	Memory	Required	Memory	Mode	Required	lines:	LMU	Memory Map
'540 progressive	720	480	D1 input	64 Mbits	2.1	61.9	-			Yes	
" "	352	240	?	"	17.7	46.3	-			-	
" "	352	480	30 fps	"	17.7	46.3	-			Yes	
" "	720	480	30 fps	"	24.0	40.0	-			Yes	
" "	352	480	60 fps	"	17.7	46.3	-		frame	-	
" "	720	480	60 fps	"	24.0	40.0	-			-	
1080 interlaced	1280	720	60 fps	"	46.8	17.2	-		frame	-	
" "	1920	1088	30 fps	"	64.0	0.0	2M/3		field	-	composite, fld
540 progressive	720	480	D1 input	96 Mbits	2.1	93.9	-			-	
" "	352	240	?	"	17.7	78.3	-				
" "	352	480	30 fps	"	17.7	78.3	-			Yes	
" "	720	480	30 fps	"	24.0	72.0	-			Yes	
" "	352	480	60 fps	"	17.7	78.3	-		frame	-	
" "	720	480	60 fps	"	24.0	72.0	-			-	
1080 interlaced	1280	720	60 fps	"	46.8	49.2	-	123.7	frame	-	
" "	1920	1088	30 fps	"	88.8	7.2	-	125.6	field	-	

Mbit= 10^6 bits

	Input Fdrmat				**						
			Frame	External			Display	OSD	LMU	Total	Free
Display Type	Hor	Vert	Rate	Memory	3 Frames	Bit Buffer	Buffer	Mem.	Mem.	Memory	Memory
540 progressive	720	480	D1 input	64	-	-		2.1	5.5	2.1	61.9
" "	352	240	?	64	6.1	9.5		2.1	-	17.7	46.3
" "	352	480	30 fps	64	6.1	9.5		2.1	2.7	17.7	46.3
" "	720	480	30 fps	64	12.4	9.5		2.1	5.5	24.0	40.0
" "	352	480	60 fps	64	6.1	9.5		2.1	-	17.7	46.3
" "	720	480	60 fps	64	12.4	9.5		2.1	-	24.0	40.0
1080 interlaced	1280	720	60 fps	64	33.2	9.5		4.1	-	46.8	17.2
" "	1920	1088	30 fps	64	50.1	9.5	0.18	4.1	-	64.0	0.0
540 progressive	720	480	D1 input	96	-	-		2.1	-	2.1	93.9
" "	352	240	?	96	6.1	9.5		2.1	-	17.7	78.3
" "	352	480	30 fps	96	6.1	9.5		2.1	2.7	17.7	78.3
" "	720	480	30 fps	96	12.4	9.5		2.1	5.5	24.0	72.0
" "	352	480	60 fps	96	6.1	9.5		2.1	-	17.7	78.3
" "	720	480	60 fps	96	12.4	9.5		2.1	-	24.0	72.0
1080 interlaced	1280	720	60 fps	96	33.2	9.5		4.1	-	46.8	49.2
" "	1920	1088	30 fps	96	75.2	9.5		4.1	-	88.8	7.2

Mbit= 10^6 bits

** Include compression

*** In H/2 Mode, uses half H res

THOMSON CONSUMER ELECTRONICS
CONFIDENTIAL AND PROPRIETARY
These drawings and specifications are the property of Thomson Consumer Electronics Inc. and shall not be reproduced or copied or used as the basis for manufacture or sale of apparatus or devices without permission.

APPENDIX Z

OVERVIEW OF IC

1.INTRODUCTION

The HD-MPEG decoder IC is targeted to be fully compliant with Grand Alliance specifications. This places certain throughput requirements on the memory used in the decoding process. These high throughput requirements can be met by Synchronous DRAM ICs. The bandwidth requirements are considered in the context of different memory bus widths, CAS latencies and clock rates. In order to guarantee fail-safe operation, the memory sub-system should be able to handle a "worst case" bitstream. From the memory point-of-view, such a bitstream would induce the greatest number of transactions on the memory bus. This in turn depends on bitstream characteristics like picture type and motion vector type. Certain non-bitstream factors also influence bus bandwidth. These include OSD data and data compression/decompression circuits.

A spreadsheet analysis is done to evaluate the effect of each contributing factor on memory bandwidth.

2.OVERVIEW

The factors influencing memory bandwidth are divided into two classes of parameters:

a) Unrestricted variables: These parameters are allowed to take on values within a specified range. Effect of changing values within specified range is incorporated in the bandwidth computations. The unrestricted variables considered are listed on page 1 of the spreadsheet

b) Restricted variables: These parameters are allowed to take on very specific values. Also, the effect of changing their values has global implications that cannot be accounted by simple formulae. So each set of restricted variables needs its own spreadsheet. Sheet 2 of the spreadsheet lists the restricted variables considered.

The choice of restricted variable values can be further narrowed down to ones that cause the greatest activity on the memory bus. The two greatest factors influencing memory bandwidth are:

a) Usage of on-the-fly compression/decompression to represent decoded frames and

b) Usage of motion compensation modes that require 4 motion predictors to decode a macroblock

Accordingly, the restricted variable values chosen for calculations lead to the following set of spreadsheets

Spreadsheet #	Restricted Variables		
	Picture type	MC mode	Compression mode
1	Frame	dual prime	two-thirds
2	Frame	bi-directional	two-thirds
3	Frame	dual-prime	M/2-H/2
4	Frame	bi-directional	M/2-H/2

Tabie i

The spreadsheet computes bandwidth requirements by dividing down all memory accesses associated with a frame, into equal portions associated with each scan line. The underlying assumption is that averaging over more than a scan line would involve unpractical amounts of on-chip RAM storage. In reality, some processing units can handle averaging over more than one scan line. Examples of these are bit buffer data, OSD data and Picture Header data.

3.MEMORY ORGANIZATION

The use of SDRAM makes it necessary to accurately model its overhead saving modes of operation. This in turn is greatly influenced by the organization of data in memory. The best memory map is one that provides the most overhead savings. The best way to save overhead is to perform paired memory accesses in which each component access is from opposing memory banks. This way overhead can be hidden even if the component accesses are from different pages of memory. In cases where component accesses are from the same page, overhead can be hidden if the number of bus words accessed is large enough to cover CAS latency. So clearly, the factors influencing overhead savings are access size and organization of data across pages and across banks. The spreadsheets are associated with the two memory map schemes shown in Fig. 2 and Fig 3. A basic 64 bit wide 64 Mbit SDRAM is organized as shown in Fig .1

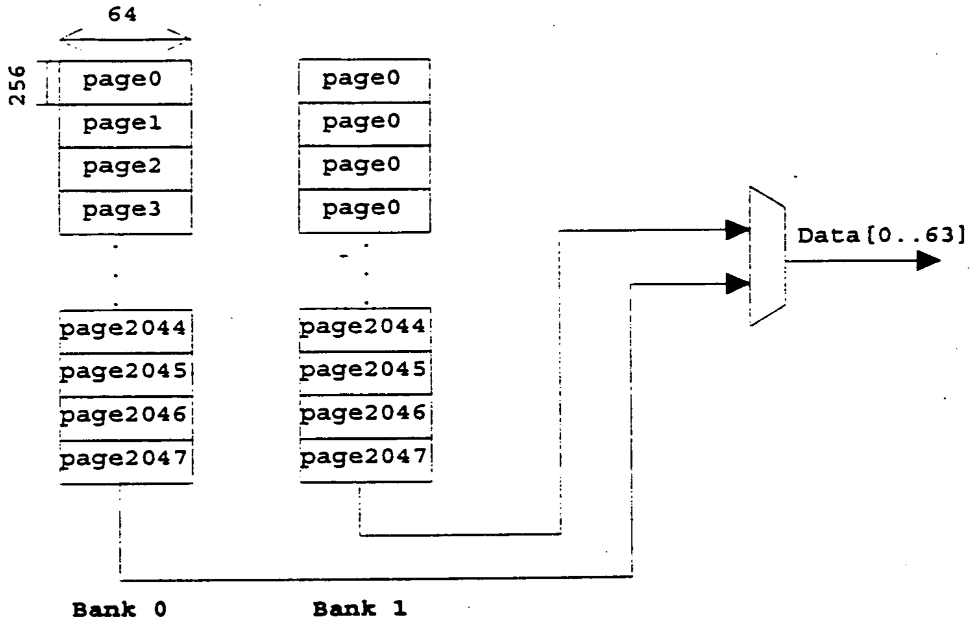


Fig.1 Basic 64 Mbit SDRAM Memory Organization

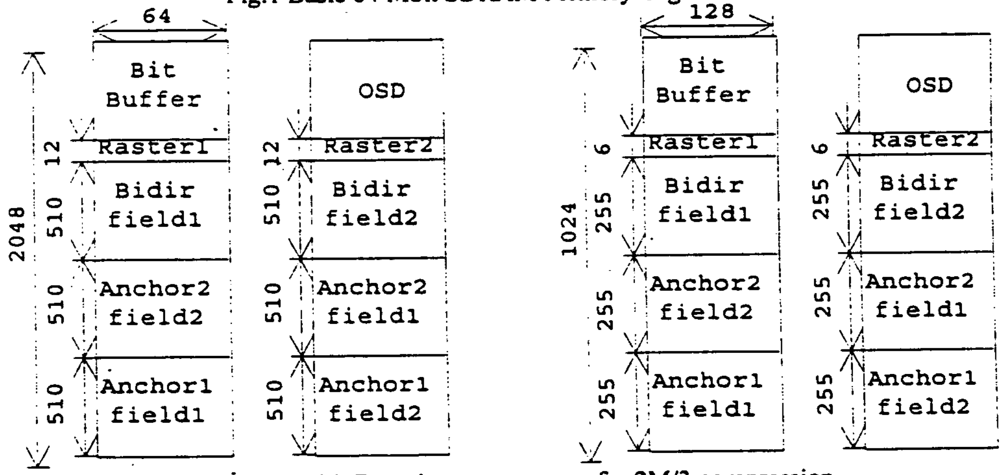


Fig. 2 64 Mbit Decoder memory map for 2M/3 compression

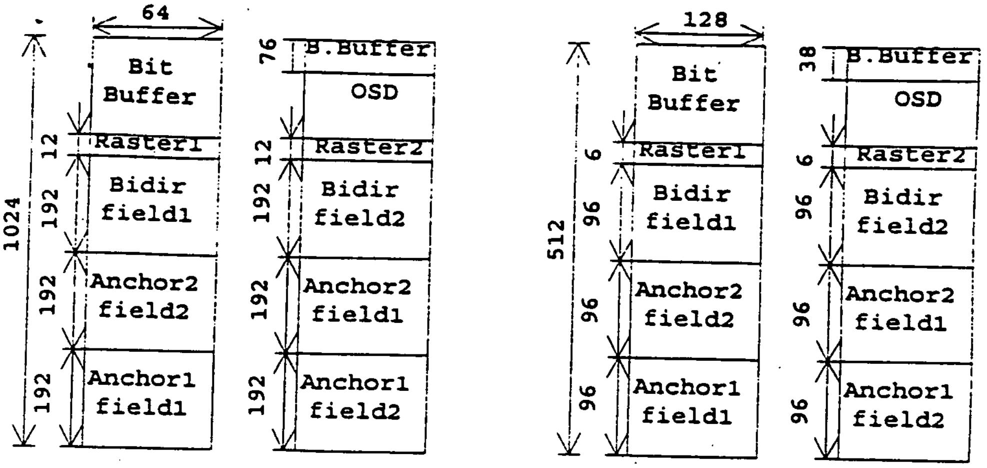


Fig 3 32 Mbit Decoder memory map for M/2:H/2 compression

4. OPERATION

The HD-MPEG decoder has three kinds of pipelines that need periodic memory access. These are, the decode pipeline, display pipeline and microprocessor pipeline. The decode pipeline is periodic on a macroblock level. That is, over each period, the following operations are performed by this pipeline:

- A piece of incoming bitstream is stored in the bit buffer space
- A piece of bitstream is read from the bit buffer into the VLD FIFO
- Up to four motion predictors are read from the two banks of memory
- Four field luma blocks (8 by 8) and eight field chroma blocks (U, V; 4 by 4) are written to the two banks of memory

The display pipeline is periodic over a scan line. During each scan line, this pipeline performs the following operations:

- A scan line of Luma and half a scan line of chroma is read out of the raster memory area of SDRAM bank currently being displayed
- An eighth of a field-wide strip of next 8 display lines of luma and chroma field blocks are read out of the field memory currently being displayed
- A scan line's worth of luma and chroma data are converted from block to scan format and written to the raster memory area of SDRAM bank currently not being displayed
- A scan line's worth of OSD data is read out of the OSD memory area
- A scan line's worth of OSD data for the next field, is written to the OSD memory area

The frame based pipeline is periodic over the VSYNC signal. Every 2 or 3 VSYNCs picture header data is sent to the microprocessor over the host bus interface. The microprocessor in turn consumes bandwidth writing to registers that control the decode and display pipelines. For the SDRAMS, refresh operations are also assumed to be periodic over VSYNC signals. This is not really necessary, but helps keep the pipelines synchronous with respect to the display operation.

Since a spreadsheet has no means of representing time, simulating an asynchronous Local Memory Controller (LMC) is not possible. So memory bandwidth is estimated, based on the assumption of synchronous operation, with the access patterns periodic over scan lines. Furthermore, the LMC is assumed to have a state machine that attempts to multiplex memory accesses in a way that best hides overhead. Keeping in view the memory maps defined in Fig. 2 and Fig. 3, following access multiplex pairs are identified:

- Field 1 predictor 1 read and Field 2 predictor 1 read (predictor frame fields are in opposite banks)
- Field 1 predictor 2 read and field 2 predictor 2 read

- Field 1 luma-chroma block #1 write and field 2 luma-chroma block #1 write (deinterlaced field components of a macroblock)
- Field 1 luma-chroma block #2 write and field 2 luma-chroma block #2 write
- Raster luma/chroma read and block-to-scan luma/chroma write (same number of pixels read and written on opposite banks of raster memory)
- OSD read and VLD read or write (Only because they are in opposite banks)
- OSD write and VLD read or write

The only unpaired memory accesses are reads of field blocks from memory to feed the block-to-scan conversion circuits. In each of these memory accesses, the basic unit of access is called a storage unit. The configuration of a storage unit depends on the size of the memory bus. But in all cases, each storage unit consists of one or more blocks of (8 by 8) luma data from one field and its associated multiplexed (4 by 4 by 2) chroma data. The spreadsheets compute worst case bandwidth requirement for this synchronous LMC by providing for the worst case scenario in terms of number of predictors and number of page hits.

5. ASSUMPTIONS

The spreadsheets are calculated with the following assumptions:

- The LMC is modeled as a device that is synchronized with the display section as detailed above
- All line stores related to sample rate conversion for luma or chroma data in the display section, are assumed to be on-chip
- Eight field lines (luma and chroma) of raster scan storage area is assumed to be available on each bank of the SDRAM for storing the results of block-to-scan conversion before these lines are displayed
- Two ping-pong buffer memory areas are assumed to be available on-chip to store decoded storage units read for the next set of 8 display lines. Each component of the ping-pong buffer is assumed to have space for at least 8 storage units so raster writes can be performed in chunks of 8 (or more) memory bus cycles (necessary to hide overhead)
- Each predictor read is split into two sets of reads that access the upper half and lower half storage units respectively. These accesses are then multiplexed with their counterparts in the second predictor of the pair. This allows the LMC to hide page hit overhead in going from the upper to the lower storage units (which are separated by a field width's worth of storage units). This is shown in Fig. 4

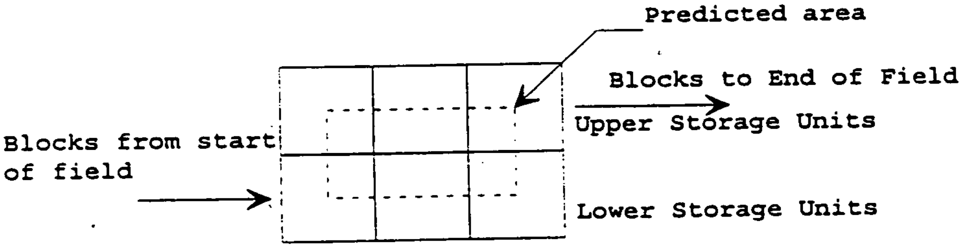


Fig. 4 Predictor read organization

HD Decoder IC memory Bandwidth estimation using NEC uPD 4516161 SDRAM

This sheet: Listing of unrestricted variables

Definitions:

Unrestricted variables: These are variable within specified range. Effects of variations are incorporated in spreadsheet

Restricted variables: Only specified values are allowed. Changing these have global implications, so each case is dealt with in its own spreadsheet

Parameter Block:		**Constants:**	
Unrestricted variables:		frame_pic_type:	0
Mem. Clock rate (MHz):	100	field_pic_type:	1
Mem. CAS Latency (clks):	3	frame_mv:	0
Input Frame Vert. Size:	1088	field_mv:	1
Input Frame Horz. Size:	1920	dual_prime:	2
1 page hit in (accesses):	4	field_16by8:	3
Input Bitrate (bits/s):	83886080	no_comp:	0
OSD Horz. Size:	1920	two_thirds:	1
OSD Vert. Size:	363	mby2_hby2:	2
OSD Mode (2 or 4 bit/pixel):	1		
OSD Resolution:	0		
Percentage of 4 mv MBs:	80	osd_mode_2:	0
Active Video Horz. Size:	1920	osd_mode_4:	1
Active Video Vert. Size:	1088	osd_res_f:	0
Total Video Horz. Size:	2400	osd_res_h:	1
Total Video Vert. Size:	1125	osd_res_t:	2
Display Frame Rate:	30		
On-chip display line stores:	0	yes:	1
SDRAM display line stores:	16	no:	0
Memory page size:	256		
External bus width (bits):	64		
Percentage of worst case MBs:	80		

```
'Function macro: Write_clks (   data_size, latency, page_hit, new_page,
'                               hide_head, hide_tail, precharge, transition  )
' Assuming use of precharge to terminate bursts
Function write_clks(data_size, latency, page_hit, new_page, hide_head, hide_tail, precharge, transition)
    page = 256
    over_head = 0
    over_tail = 0
    body = 0

    If hide_head = 0 Then
        'hide_head will be FALSE (=0) for read -> write transitions from
        'the same bank when burst mode is page.  So page mode read->write
        'transition on same bank will look like non-overlapping accesses.
        'This was made necessary because read and write routines don't
        'include bank switching as part of overhead calculations.  They
        'rely on the hide_head and hide_tail variables to provide this
        'mechanism
        If precharge = 1 Then
            over_head = over_head + 3   'precharge to activation overhead
        End If
        If new_page = 1 Then
            'A new page is deemed to occur everytime the LMU reads from
            'a new area of mem. or if the LMU was forced to terminate
            'the last burst of data using a precharge (in the absence of
            'a pending access request)
            over_head = over_head + 3   'Activation overhead
        End If
    ElseIf transition = 1 Then
        'Following applies to all burst modes and latencies except page
        'mode, where it applies only in cases where the two accesses are
        'from different banks
        over_head = over_head + 1   'read to write Hi-Z cycle overhead
    End If
    If hide_tail = 0 Then
        If latency = 1 Or latency = 2 Then
            over_tail = over_tail + 1 + 3   'last data to precharge to activation
        Else
            over_tail = over_tail + 2 + 3
        End If
    End If
    If page_hit = 1 Then
        'We are ignoring the potential for the LMU to hide this
        'overhead by servicing a request from the other bank
        If latency = 1 Or latency = 2 Then
            body = body + 1 + 3 + 3
        Else
```

```
            body = body + 2 + 3 + 3
        End If
    End If

    write_clks = over_head + over_tail + body + data_size

End Function
```

```
'Function macro: Read_clks (     data_size, latency, page_hit, new_page,
'                                hide_head, hide_tail, precharge, transition  )
' Assuming use of precharge to terminate bursts
Function read_clks(data_size, latency, page_hit, new_page, hide_head, hide_tail, precharge, transition)
    page = 256
    over_head = 0
    over_tail = 0
    body = 0

    If hide_head = 0 Then
        If precharge = 1 Then
            over_head = over_head + 3   'precharge to activation overhead
        End If
        If new_page = 1 Then
            'A new page is deemed to occur everytime the LMU reads from
            'a new area of mem. or if the LMU was forced to terminate
            'the last burst of data using a precharge (in the absence of
            'a pending access request)
            over_head = over_head + 3   'Activation overhead
        End If
        over_head = over_head + latency 'Also applies to write->read xsition
    ElseIf transition = 1 Then
        over_head = over_head + 1       'write to read Hi-Z cycle overhead
    End If
    If hide_tail = 0 Then
        over_tail = over_tail + 3
    End If
    If page_hit = 1 Then
        'We are ignoring the potential for the LMU to hide this
        'overhead by servicing a request from the other bank
        body = body + 3 + 3 + latency
    End If

    read_clks = over_head + over_tail + body + data_size

End Function
```

```
'Function macro: stu_per_mb(    bus_width    )
'
'This function computes the number of storage units in each macroblock
'The bus_widths supported for these calculations are 64 and 128 only.
'bus_width takes values 64 and 128
Function stu_per_mb(b_width)
    Select Case b_width
    Case 64
        stu_per_mb = 4
    Case 128
        stu_per_mb = 2
    End Select

End Function
```

```
'Function macro: block_clks(    comp_mode, bus_width    )
'
'This function computes the number of cycles required to access a storage
'unit from memory.  The definition of a storage unit depends on the
'bus_width parameter.  The bus_widths supported for these calculations are
'64 and 128 only.  For 64 bit bus, the storage unit is one field-block.  For
'128 bit bus, it is two adjacent field blocks.
'comp_mode takes values 0, 1, 2 corresponding to no compression, two-thirds
'and m/2;h/2
'bus_width takes values 64 and 128
Function block_clks(c_mode, b_width)
    Select Case b_width
    Case 64
        luma_pix = 64
        chroma_pix = 16
    Case 128
        luma_pix = 128
        chroma_pix = 32
    End Select

    Select Case c_mode
    Case 0
        'no compression
        block_clks = (luma_pix * 8) / b_width + (chroma_pix * 2 * 8) / b_width
    Case 1
        'two-thirds compression
        block_clks = (luma_pix * 8 * 3) / (4 * b_width) + (chroma_pix * 2 * 8) / (2 * b_width)
    Case 2
        'M/2-H/2 compression
        block_clks = (luma_pix * 8) / (b_width * 4) + (chroma_pix * 2 * 8) / (b_width * 4)
    End Select

End Function
```

```
'Function macro: blk2ras_dec(   stu_clks, latency, stu_per_pg, stu_per_ln, repeat_cnt  )
'
'This function computes the number of cycles required to decode stu_per_ln
'number of field blocks.  These field blocks form the next 8 field lines
'that will be displayed.  The variable stu_per_pg helps calculates the worst
'case number of page hits incurred in reading stu_per_ln field blocks.  The
'variable repeat_cnt is the number of times the basic memory access pattern
'is repeated during a line.  The variable stu_clks is the number of mem. cycles
'required to access one storage unit.  The variable latency is the SDRAM CAS
'latency

Function blk2ras_dec(stu_clks, latency, stu_per_pg, stu_per_ln, repeat_cnt)
    If (Int((stu_per_ln / stu_per_pg)) < (stu_per_ln / stu_per_pg) Then
        num_pg_hits = Int((stu_per_ln / stu_per_pg)) + 1
    Else
        num_pg_hits = stu_per_ln / stu_per_pg
    End If
    If (Int((stu_per_ln / repeat_cnt)) < (stu_per_ln / repeat_cnt) Then
        blks_per_cycle = Int((stu_per_ln / repeat_cnt)) + 1
    Else
        blks_per_cycle = stu_per_ln / repeat_cnt
    End If
    clk_cnt = 0
    blk_cnt = 0
    For cyc_count = 1 To repeat_cnt Step 1
        If (blk_cnt + blks_per_cycle) <= stu_per_ln Then
            If num_pg_hits > 0 Then
                'Very first access in a cycle is new page
                clk_cnt = clk_cnt + read_clks(stu_clks, latency, 0, 1, 0, 0, 1, 0)
                'Assuming blks_per_cycle does not span more than 1 page!
                clk_cnt = clk_cnt + read_clks(stu_clks, latency, 0, 1, 0, 1, 1, 0)
                num_pg_hits = num_pg_hits - 1
                If blks_per_cycle > 2 Then
                    clk_cnt = clk_cnt + read_clks((stu_clks * (blks_per_cycle - 2)), latency, 0, 0, 1, 1, 0, 0)
                End If
            Else
                'Very first access in a cycle is new page
                clk_cnt = clk_cnt + read_clks(stu_clks, latency, 0, 1, 0, 1, 1, 0)
                clk_cnt = clk_cnt + read_clks((stu_clks * (blks_per_cycle - 1)), latency, 0, 0, 1, 1, 0, 0)
            End If
            blk_cnt = blk_cnt + blks_per_cycle
        Else
            If num_pg_hits > 0 Then
                'Very first access in a cycle is new page
                clk_cnt = clk_cnt + read_clks(stu_clks, latency, 0, 1, 0, 0, 1, 0)
                blk_cnt = blk_cnt + 1
                clk_cnt = clk_cnt + read_clks(stu_clks, latency, 0, 1, 0, 1, 1, 0)
                num_pg_hits = num_pg_hits - 1
                blk_cnt = blk_cnt + 1
            Else
                'Very first access in a cycle is new page
                clk_cnt = clk_cnt + read_clks(stu_clks, latency, 0, 1, 0, 1, 1, 0)
                blk_cnt = blk_cnt + 1
            End If
            If blk_cnt < stu_per_ln Then
                clk_cnt = clk_cnt + read_clks((stu_clks * (stu_per_ln - blk_cnt)), latency, 0, 0, 1, 1, 0, 0)
                blk_cnt = stu_per_ln
            End If
        End If
    Next cyc_count
    blk2ras_dec = clk_cnt
End Function
```

```
'Function macro: osd_clks(  b_width, h_size, v_size, mode, res, frame_lines )
'
'This function computes the number of memory cycles required per
'scan line to satisfy reading or writing of OSD information from
'or to the SDRAM.  Variable b_width denotes the SDRAM bus bandwidth
'Variables h_size and v_size represent the worst case horizontal and
'vertical sizes of OSD.  Variable mode refers to 2 bit/pixel (=0) or
'4 bit/pixel (1) OSD.  Variable res refers to OSD resolution: 0=>full
';1=>half and 2=>one-third.  Variable frame_lines refers to the number
'of lines between two odd synchs

Function osd_clks(b_width, h_size, v_size, mode, res, frame_lines)
    pel_area = h_size * v_size
    Select Case b_width
    Case 64
        Select Case mode
        Case 0
            Select Case res
            Case 0
                    frame_clks = pel_area / 32
            Case 1
                    frame_clks = pel_area / 64
            Case 2
                    frame_clks = pel_area / 96
            End Select
        Case 1
            Select Case res
            Case 0
                    frame_clks = pel_area / 16
            Case 1
                    frame_clks = pel_area / 32
            Case 2
                    frame_clks = pel_area / 48
            End Select
        End Select
    Case 128
        Select Case mode
        Case 0
            Select Case res
            Case 0
                    frame_clks = pel_area / 64
            Case 1
                    frame_clks = pel_area / 128
            Case 2
                    frame_clks = pel_area / 192
            End Select
        Case 1
            Select Case res
            Case 0
                    frame_clks = pel_area / 32
            Case 1
                    frame_clks = pel_area / 64
            Case 2
                    frame_clks = pel_area / 96
            End Select
        End Select
    End Select
    If (Int(frame_clks / frame_lines)) < (frame_clks / frame_lines) Then
        osd_clks = Int(frame_clks / frame_lines) + 1
    Else
        osd_clks = frame_clks / frame_lines
    End If
End Function
```

```
'Function macro: raster_rd_wr( act_ln_pels, bus_width, stu_per_ln, latency )
'
'This function computes the number of cycles required to perform paired
'read-write operations of 4:2:0 format display data.  Each line, a raster
'line of display luma and half a raster line of display chroma are read
'out of one bank, while the same number of luma and chroma pels are written
'into the opposite bank from the display section decode of stu_per_ln
'storage units.  The variable act_ln_pels represents the number of active luma
'pels in each line.  Variables bus_width and latency denote SDRAM bus width and
'CAS latency.

Function raster_rd_wr(act_ln_pels, bus_width, stu_per_ln, latency)

    clk_cnt = 0
    write_cnt = 0
    read_cnt = 0
    rd_pel_cnt = 0
    wr_pel_cnt = 0
    Select Case bus_width
    Case 64
        'Each storage unit is a field block - an 8 by 8 luma block with 2 4 by 4
        'chroma blocks
        'No matter what the latency, PRE-ACTIVE (3 clks) and ACTIVE-READ/WRITE
        '(3 clks) will mandate 8 access cycles for zero overhead bank mux'ed
        'operations.  The paired bank mux'ed operations should escape from first
        'access and last access overhead if the LMC algorithm is synchronous enough
        'Active line pels is always a multiple of 16
        While ((write_cnt + 8) <= stu_per_ln) And ((read_cnt + 8) <= (Int(act_ln_pels / 512)))
            'Assume worst case, that every pair induces a read<->write transition
            For luma_blk = 1 To 8 Step 1
                clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 1, 1, 1, 1)
                clk_cnt = clk_cnt + write_clks(8, latency, 0, 1, 1, 1, 1, 1)
            Next luma_blk
            For chroma_blk = 1 To 4 Step 1
                clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 1, 1, 1, 1)
                clk_cnt = clk_cnt + write_clks(8, latency, 0, 1, 1, 1, 1, 1)
            Next chroma_blk
            write_cnt = write_cnt + 8
            read_cnt = read_cnt + 8
            rd_pel_cnt = rd_pel_cnt + 512
            wr_pel_cnt = wr_pel_cnt + 512
        Wend
        While ((rd_pel_cnt + 128) <= act_ln_pels) And ((wr_pel_cnt + ((stu_per_ln - write_cnt) * 16)) <= (stu_per_
ln * 64))
            'Two luma accesses and ...
            clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 1, 1, 1, 1)
            clk_cnt = clk_cnt + write_clks((stu_per_ln - write_cnt), latency, 0, 1, 1, 0, 1, 1)
            clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 1, 1, 1, 1)
            clk_cnt = clk_cnt + write_clks((stu_per_ln - write_cnt), latency, 0, 1, 1, 0, 1, 1)
            '... one chroma access
            clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 1, 1, 1, 1)
            clk_cnt = clk_cnt + write_clks((stu_per_ln - write_cnt), latency, 0, 1, 1, 0, 1, 1)
            rd_pel_cnt = rd_pel_cnt + 128
            wr_pel_cnt = wr_pel_cnt + (stu_per_ln - write_cnt) * 16
        Wend
        If rd_pel_cnt < act_ln_pels Then
            clk_cnt = clk_cnt + read_clks(((act_ln_pels - rd_pel_cnt) / 8), latency, 0, 1, 0, 1, 1, 1)
            clk_cnt = clk_cnt + read_clks(((act_ln_pels - rd_pel_cnt) / 16), latency, 0, 1, 0, 1, 1, 1)
        End If
        While wr_pel_cnt < (stu_per_ln * 64)
            'Two luma field block lines and ...
            clk_cnt = clk_cnt + write_clks(((stu_per_ln - write_cnt) * 2), latency, 0, 1, 0, 1, 1, 1)
            '...one chroma field block line
            clk_cnt = clk_cnt + write_clks((stu_per_ln - write_cnt), latency, 0, 1, 0, 1, 1, 1)
            wr_pel_cnt = wr_pel_cnt + (stu_per_ln - write_cnt) * 16
        Wend

    Case 128
        'Each storage unit is two horizontally adjascent field blocks
        'No matter what the latency, PRE-ACTIVE (3 clks) and ACTIVE-READ/WRITE
        '(3 clks) will mandate 8 access cycles for zero overhead bank mux'ed
        'operations.  The paired bank mux'ed operations should escape from first
        'access and last access overhead if the LMC algorithm is synchronous enough
        'Active line pels is always a multiple of 16
        While ((write_cnt + 8) <= stu_per_ln) And ((read_cnt + 8) <= (Int(act_ln_pels / 1024)))
            'Assume worst case, that every pair induces a read<->write transition
            For luma_blk = 1 To 8 Step 1
                clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 1, 1, 1, 1)
                clk_cnt = clk_cnt + write_clks(8, latency, 0, 1, 1, 1, 1, 1)
            Next luma_blk
            For chroma_blk = 1 To 4 Step 1
                clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 1, 1, 1, 1)
                clk_cnt = clk_cnt + write_clks(8, latency, 0, 1, 1, 1, 1, 1)
            Next chroma_blk
            write_cnt = write_cnt + 8
            read_cnt = read_cnt + 8
            rd_pel_cnt = rd_pel_cnt + 1024
            wr_pel_cnt = wr_pel_cnt + 1024
        Wend
        While ((rd_pel_cnt + 256) <= act_ln_pels) And ((wr_pel_cnt + ((stu_per_ln - write_cnt) * 32)) <= (stu_per_
ln * 128))
            'Two luma accesses and ...
            clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 1, 1, 1, 1)
            clk_cnt = clk_cnt + write_clks((stu_per_ln - write_cnt), latency, 0, 1, 1, 0, 1, 1)
            clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 1, 1, 1, 1)
            clk_cnt = clk_cnt + write_clks((stu_per_ln - write_cnt), latency, 0, 1, 1, 0, 1, 1)
            '... one chroma access
            clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 1, 1, 1, 1)
            clk_cnt = clk_cnt + write_clks((stu_per_ln - write_cnt), latency, 0, 1, 1, 0, 1, 1)
            rd_pel_cnt = rd_pel_cnt + 256
            wr_pel_cnt = wr_pel_cnt + (stu_per_ln - write_cnt) * 32
        Wend
        If rd_pel_cnt < act_ln_pels Then
            clk_cnt = clk_cnt + read_clks(((act_ln_pels - rd_pel_cnt) / 16), latency, 0, 1, 0, 1, 1, 1)
            clk_cnt = clk_cnt + read_clks(((act_ln_pels - rd_pel_cnt) / 32), latency, 0, 1, 0, 1, 1, 1)
        End If
        While wr_pel_cnt < (stu_per_ln * 128)
            'Two luma field block lines and ...
            clk_cnt = clk_cnt + write_clks(((stu_per_ln - write_cnt) * 2), latency, 0, 1, 0, 1, 1, 1)
            '...one chroma field block line
            clk_cnt = clk_cnt + write_clks((stu_per_ln - write_cnt), latency, 0, 1, 0, 1, 1, 1)
            wr_pel_cnt = wr_pel_cnt + (stu_per_ln - write_cnt) * 32
        Wend
    End Select
    raster_rd_wr = clk_cnt
End Function
```

```
'Function macro: vld_clks(  b_width, bitrate, frame_rate, frame_lines   )
'
'This function computes the number of memory cycles required per
'scan line to satisfy reading or writing of compressed bitstream
'from or to the bit buffer

Function vld_clks(b_width, bitrate, frame_rate, frame_lines)

    If (Int(bitrate / (b_width * frame_rate * frame_lines))) < (bitrate / (b_width * frame_rate * frame_lines)) Th
en
        vld_clks = Int(bitrate / (b_width * frame_rate * frame_lines)) + 1
    Else
        vld_clks = bitrate / (b_width * frame_rate * frame_lines)
    End If

End Function
```

```
'Function macro: osd_vld_rd_wr(  osd_per_ln, vld_per_ln, latency    )
'
'This function computes the number of cycles required to perform paired
'read-write operations of OSD and VLD data.  Each line, a portion of data from
'a bitstream input FIFO is transferred to SDRAM.  At the same time, an OSD
'input FIFO is also serviced if data is available.  The two transfers occur in
'opposite memory banks.  So overhead can be hidden, provided each session is 8
'bus cycles long.  Variables osd_per_ln and vld_per_ln denote the number of OSD
'and VLD read and write cycles that are needed per scan line.  Variable latency
'denotes SDRAM CAS latency.

Function osd_vld_rd_wr(osd_per_ln, vld_per_ln, latency)

    clk_cnt = 0
    osd_rd_cnt = 0
    osd_wr_cnt = 0
    vld_rd_cnt = 0
    vld_wr_cnt = 0

    'No matter what the latency, PRE-ACTIVE (3 clks) and ACTIVE-READ/WRITE
    '(3 clks) will mandate 8 access cycles for zero overhead bank mux'ed
    'operations.  The paired bank mux'ed operations should escape from first
    'access and last access overhead if the LMC algorithm is synchronous enough
    'First access to account for page hit overhead
    If (osd_per_ln > 8) Then
        clk_cnt = clk_cnt + read_clks(8, latency, 1, 0, 1, 1, 1, 1)
        clk_cnt = clk_cnt + write_clks(8, latency, 1, 0, 1, 1, 1, 1)
        osd_rd_cnt = osd_rd_cnt + 8
        osd_wr_cnt = osd_wr_cnt + 8
    Else
        clk_cnt = clk_cnt + read_clks(osd_per_ln, latency, 1, 0, 0, 1, 1, 1)
        clk_cnt = clk_cnt + write_clks(osd_per_ln, latency, 1, 0, 0, 1, 1, 1)
        osd_rd_cnt = osd_per_ln
        osd_wr_cnt = osd_per_ln
    End If
    If (vld_per_ln > 8) Then
        clk_cnt = clk_cnt + read_clks(8, latency, 1, 0, 1, 1, 1, 0)
        clk_cnt = clk_cnt + write_clks(8, latency, 1, 0, 1, 1, 1, 0)
        vld_rd_cnt = vld_rd_cnt + 8
        vld_wr_cnt = vld_wr_cnt + 8
    Else
        clk_cnt = clk_cnt + read_clks(vld_per_ln, latency, 1, 0, 0, 1, 1, 0)
        clk_cnt = clk_cnt + write_clks(vld_per_ln, latency, 1, 0, 0, 1, 1, 0)
        vld_rd_cnt = vld_per_ln
        vld_wr_cnt = vld_per_ln
    End If
```

```
While ((osd_rd_cnt + 8) <= osd_per_ln) And ((vld_rd_cnt + 8) <= vld_per_ln)
     'Worst case both reads are inserted between two writes.  But we assume
     'more intelligence on the part of the LMC.   Note that this worst case
     'situation cannot be represented in this model as the trailing write has
     'no idea about these read operations
    clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 1, 1, 1, 1)
    clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 1, 1, 1, 0)
    osd_rd_cnt = osd_rd_cnt + 8
    vld_rd_cnt = vld_rd_cnt + 8
Wend
While ((osd_wr_cnt + 8) <= osd_per_ln) And ((vld_wr_cnt + 8) <= vld_per_ln)
     'See comments above
    clk_cnt = clk_cnt + write_clks(8, latency, 0, 1, 1, 1, 1, 1)
    clk_cnt = clk_cnt + write_clks(8, latency, 0, 1, 1, 1, 1, 0)
    osd_wr_cnt = osd_wr_cnt + 8
    vld_wr_cnt = vld_wr_cnt + 8
Wend
While ((osd_wr_cnt + 8) <= osd_per_ln) And ((vld_rd_cnt + 8) <= vld_per_ln)
    clk_cnt = clk_cnt + write_clks(8, latency, 0, 1, 1, 1, 1, 1)
    clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 1, 1, 1, 1)
    osd_wr_cnt = osd_wr_cnt + 8
    vld_rd_cnt = vld_rd_cnt + 8
Wend
While ((osd_rd_cnt + 8) <= osd_per_ln) And ((vld_wr_cnt + 8) <= vld_per_ln)
    clk_cnt = clk_cnt + write_clks(8, latency, 0, 1, 1, 1, 1, 1)
    clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 1, 1, 1, 1)
    osd_rd_cnt = osd_rd_cnt + 8
    vld_wr_cnt = vld_wr_cnt + 8
Wend
While (osd_rd_cnt + 8) <= osd_per_ln
    clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 0, 1, 1, 0)
    osd_rd_cnt = osd_rd_cnt + 8
Wend
While (osd_wr_cnt + 8) <= osd_per_ln
    clk_cnt = clk_cnt + write_clks(8, latency, 0, 1, 0, 1, 1, 0)
    osd_wr_cnt = osd_wr_cnt + 8
Wend
While (vld_rd_cnt + 8) <= vld_per_ln
    clk_cnt = clk_cnt + read_clks(8, latency, 0, 1, 0, 1, 1, 0)
    vld_rd_cnt = vld_rd_cnt + 8
Wend
While (vld_wr_cnt + 8) <= vld_per_ln
    clk_cnt = clk_cnt + write_clks(8, latency, 0, 1, 0, 1, 1, 0)
    vld_wr_cnt = vld_wr_cnt + 8
Wend
If osd_rd_cnt < osd_per_ln Then
```

```
        clk_cnt = clk_cnt + read_clks((osd_per_ln - osd_rd_cnt), latency, 0, 1, 0, 1, 1, 0)
    End If
    If osd_wr_cnt < osd_per_ln Then
        clk_cnt = clk_cnt + write_clks((osd_per_ln - osd_wr_cnt), latency, 0, 1, 0, 1, 1, 0)
    End If
    If vld_rd_cnt < vld_per_ln Then
        clk_cnt = clk_cnt + read_clks((vld_per_ln - vld_rd_cnt), latency, 0, 1, 0, 1, 1, 0)
    End If
    If vld_wr_cnt < vld_per_ln Then
        clk_cnt = clk_cnt + write_clks((vld_per_ln - vld_wr_cnt), latency, 0, 1, 0, 1, 1, 0)
    End If

    osd_vld_rd_wr = clk_cnt

End Function
```

```
'Function macro: predictor_clks(    b_width, latency, stu_clks, mbs_per_ln  )
'
'This function computes the number of clock cycles required per H reset
'to read in 4 (worst case) predictors from memory, for each macroblock
'processed in that interval.  Variables b_width and latency denote the
'bus width and CAS latency of the target SDRAM.  Variable stu_clks
'represents the number of memory cycles required to access one storage
'unit in memory.  Variable mbs_per_ln represents the number of
'macroblocks processed per H reset

Function predictor_clks(b_width, latency, stu_clks, mbs_per_ln)

    clk_cnt = 0

    Select Case b_width
    Case 64
        'A storage unit is one 8 by 8 field block.  Six such st. units are
        'accessed to form one predictor.  Each predictor, given the memory
        'organization, can have at most one page crossing (in the upper 3
        'or lower 3 blocks but not both at the same time)

        'First macroblock in scan line ...
        'First field upper half ...
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 1, 1, 0, 1, 1, 1)
        '... Second field upper half ...
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 1, 1, 1, 1, 1, 0)
        '... First field lower half ...
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 1, 1, 1, 0)
        '... Second field lower half ...
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 1, 1, 1, 0)
        '... and similarly for the other two predictors:
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 1, 1, 1, 1, 1, 0)
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 1, 1, 1, 1, 1, 0)
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 1, 1, 1, 0)
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 1, 1, 1, 0)
        'Writing 4 storage units to memory:
        'First writes in each field ...
        clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 1, 1, 1, 1)
        clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 1, 1, 1, 0)
        'Second writes in each field with page crossing (need precharge)
        clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 1, 1, 1, 0)
        clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 1, 1, 1, 0)

        'Intermediete macroblocks in scan line ....
        For i = 1 To (mbs_per_ln - 2) Step 1
            'First field upper half ...
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 1, 1, 1, 1, 1, 1)
        '... Second field upper half ...
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 1, 1, 1, 1, 1, 0)
        '... First field lower half ...
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 1, 1, 1, 0)
        '... Second field lower half ...
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 1, 1, 1, 0)
        '... and similarly for the other two predictors:
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 1, 1, 1, 1, 1, 0)
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 1, 1, 1, 1, 1, 0)
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 1, 1, 1, 0)
        clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 1, 1, 1, 0)
        'Writing 4 storage units to memory:
        'First writes in each field ...
        clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 1, 1, 1, 1)
        clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 1, 1, 1, 0)
        'Second writes in each field without page crossing (no precharge)
        clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 0, 1, 1, 0, 0)
        clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 0, 1, 1, 0, 0)
    Next i
    'Last macroblock in scan line
    'First field upper half ...
    clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 1, 1, 1, 1, 1, 1)
    '... Second field upper half ...
    clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 1, 1, 1, 1, 1, 0)
    '... First field lower half ...
    clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 1, 1, 1, 0)
    '... Second field lower half ...
    clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 1, 1, 1, 0)
    '... and similarly for the other two predictors:
    clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 1, 1, 1, 1, 1, 0)
    clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 1, 1, 1, 1, 1, 0)
    clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 1, 1, 1, 0)
    clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 1, 1, 1, 0)
    'Writing 4 storage units to memory:
    'First writes in each field ...
    clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 1, 1, 1, 1)
    clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 1, 1, 1, 0)
    'Second writes in each field without page crossing (no precharge)
    clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 0, 1, 1, 0, 0)
    clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 0, 1, 0, 0, 0)
Case 128
    'A storage.unit is 2 adjascent 8 by 8 field blocks.  Four such st. units
    'are accessed to form one predictor.  Each predictor, given the mem.
    'organization, can have at most one page crossing (in the upper 2 units
    'or the lower 2 units but not both at the same time)
```

```
'First macroblock in scan line ...
'First field upper half ...
clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 0, 1, 1, 1)
'... Second field upper half ...
clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 1, 1, 1, 0)
'... First field lower half ...
clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 1, 1, 1, 0)
'... Second field lower half ...
clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 1, 1, 1, 0)
'... and similarly for the other two predictors:
clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 1, 1, 1, 0)
clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 1, 1, 1, 0)
clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 1, 1, 1, 0)
clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 1, 1, 1, 0)
'Writing 2 storage units to memory:
clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 1, 1, 1, 1)
clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 1, 1, 1, 0)

'Intermediete macroblock in scan line ...
For i = 1 To (mbs_per_ln - 2) Step 1
    'First field upper half ...
    clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 1, 1, 1, 1)
    '... Second field upper half ...
    clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 1, 1, 1, 0)
    '... First field lower half ...
    clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 1, 1, 1, 0)
    '... Second field lower half ...
    clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 1, 1, 1, 0)
    '... and similarly for the other two predictors:
    clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 1, 1, 1, 0)
    clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 1, 1, 1, 0)
    clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 1, 1, 1, 0)
    clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 1, 1, 1, 0)
    'Writing 2 storage units to memory:
    clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 1, 1, 1, 1)
    clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 1, 1, 1, 0)
Next i

'And the last macroblock
'First field upper half ....
clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 1, 1, 1, 1)
'... Second field upper half ...
clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 1, 1, 1, 0)
'... First field lower half ...
clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 1, 1, 1, 0)
'... Second field lower half ...
```

```
        clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 1, 1, 1, 0)
        '... and similarly for the other two predictors:
        clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 1, 1, 1, 0)
        clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 1, 1, 1, 0)
        clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 1, 1, 1, 0)
        clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 1, 1, 1, 0)
        'Writing 2 storage units to memory:
        clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 1, 1, 1, 1)
        clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 1, 0, 1, 0)
    End Select

    predictor_clks = clk_cnt

End Function
```

HD Decoder IC memory Bandwidth estimation using NEC uPD 4516161 SDRAM

This sheet: Computation of memory bandwidth loads for one set of restricted variables

Assumptions:

1) Each reconstructed frame is stored as two independent fields in memory
2) Luma-chroma data from one field in a macroblock is stored as a unit called field-macroblock
3) Reconstructed storage units in memory do not cross page boundaries
4)Each reconstructed field memory area starts at a page boundary

Restricted variables:		Worst case Scenario:	
		1) All macroblocks processed in a line are coded with dual prime vectors	
Picture type (0=>frame):	0	2) OSD information is being updated with page hit	
Motion Estimation mode:	2	3) OSD information is being read for display with page hit	
Compression mode:	1	4) One word of PES header data is read out	
		5) Every set of predictors crosses a page boundary	
		6) Compressed bitstream is being read out with page hit	
		7) Compressed bitstream is being written with page hit	
Tot. MBs in input frame:	8160	OSD read or update cycles per H reset:	39
Frame time (ns):	33333333	Bit buffer read or write cycles per H reset:	39
MBs processed per H_reset:	8	OSD & VLD read-write cycles per H reset:	228
Storage Unit access clocks:	8	PES Header read cycles per H reset:	20
Storage Units per macroblock:	4	Refresh overhead per field:	2048
Storage Units per page:	32		
		Sum total read & write cycles per H reset:	3503
		Per field total bandwidth including refresh:	1972486
		Memory bus clock frequency:	118349160
Macroblock only cycles per H reset:	2109	Bandwidth calculations assuming on-chip block-to-scan conversion:	
Storage Units for display per H reset:	30	Sum total read & write cycles per H reset:	2681
Display decoder read cycles per H reset:	324	Per field total bandwidth including refresh:	1510111
Display line buffer read and write:	822	Memory bus clock frequency:	[redacted]

```
    '... First field lower half ...
    clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 1, 1, 1, 0)
    '... Second field lower half ...
    clk_cnt = clk_cnt + read_clks((3 * stu_clks), latency, 0, 1, 1, 1, 1, 0)

    'Writing 4 storage units to memory:
    'First writes in each field ...
    clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 1, 1, 1, 1)
    clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 0, 1, 1, 0)
    'Second writes in each field without page crossing (no precharge)
    clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 0, 0, 1, 0, 0)
    clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 0, 0, 0, 0, 0)
Case 128
    'A storage unit is 2 adjascent 8 by 8 field blocks.  Four such st. units
    'are accessed to form one predictor.  Each predictor, given the mem.
    'organization, can have at most one page crossing (in the upper 2 units
    'or the lower 2 units but not both at the same time)
    'Number of read bus cycles are not large enough to fully hide overhead
    'Note:  Tail overhead gives a more meaningful number of overhead cycles
    '       This is not representative of what is assumed to be happening
    'First macroblock in scan line ...
    'First field upper half ...
    clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 0, 1, 1, 1)
    '... Second field upper half ...
    clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 1, 0, 1, 0)
    '... First field lower half ...
    clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 1, 0, 1, 0)
    '... Second field lower half ...
    clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 1, 0, 1, 0)

    'Writing 2 storage units to memory:
    'Writes are not large enough to hide overhead
    clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 1, 1, 1, 1)
    clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 0, 1, 1, 0)

    'Intermediete macroblock in scan line ...
    For i = 1 To (mbs_per_ln - 2) Step 1
        'First field upper half ...
        clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 0, 1, 1, 1)
        '... Second field upper half ...
        clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 1, 0, 1, 0)
        '... First field lower half ...
        clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 1, 0, 1, 0)
        '... Second field lower half ...
        clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 1, 0, 1, 0)

        'Writing 2 storage units to memory:
        clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 1, 1, 1, 1)
        clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 0, 1, 1, 0)
    Next i

    'And the last macroblock
    'First field upper half ...
    clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 0, 1, 1, 1)
    '... Second field upper half ...
    clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 1, 1, 1, 0, 1, 0)
    '... First field lower half ...
    clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 1, 0, 1, 0)
    '... Second field lower half ...
    clk_cnt = clk_cnt + read_clks((2 * stu_clks), latency, 0, 1, 1, 0, 1, 0)

    'Writing 2 storage units to memory:
    clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 1, 1, 1, 1)
    clk_cnt = clk_cnt + write_clks(stu_clks, latency, 0, 1, 0, 0, 1, 0)
End Select
```

```
    pred_bc_clks = clk_cnt
End Function
```

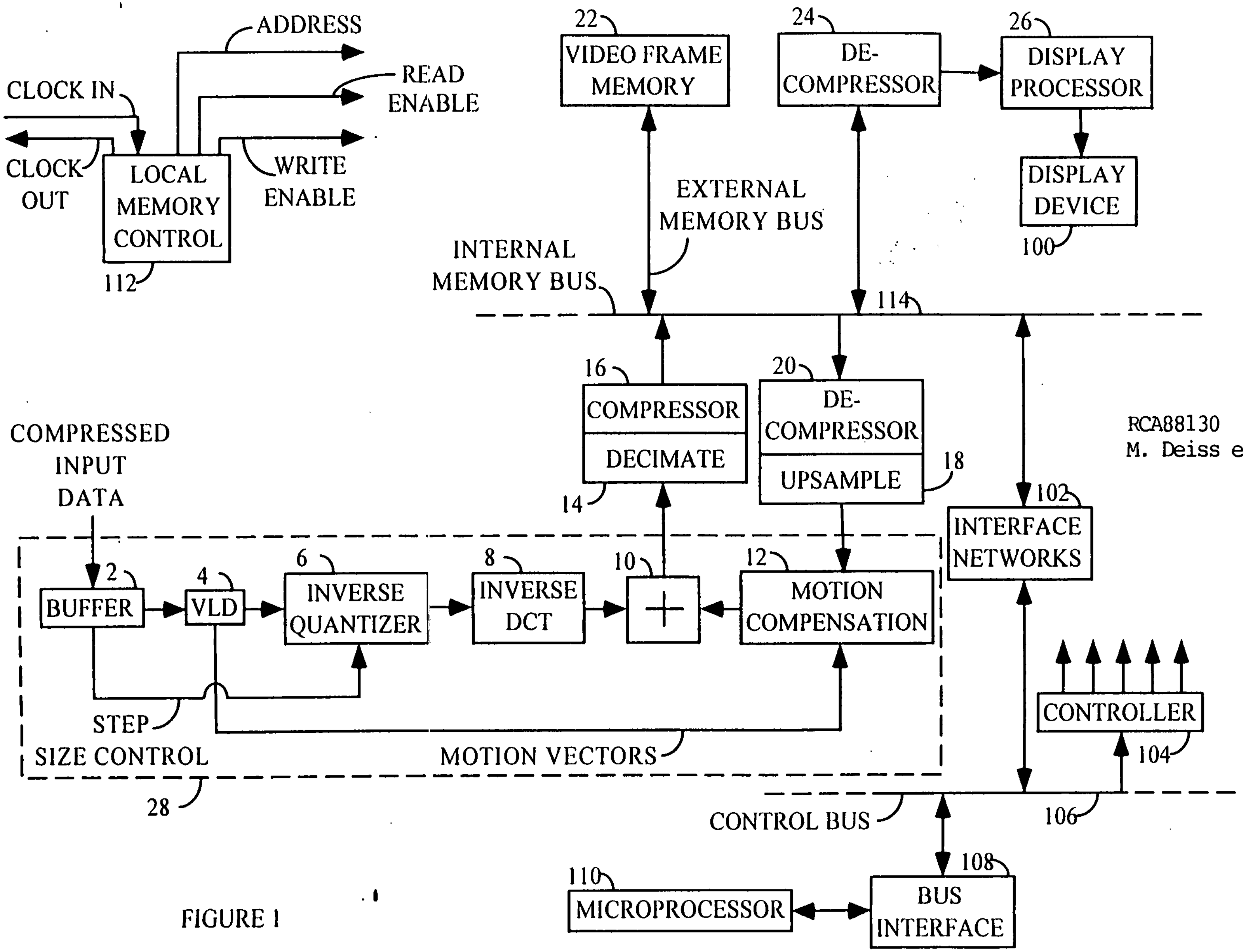
HD Decoder IC memory Bandwidth estimation using NEC uPD 4516161 SDRAM

This sheet: Computation of memory bandwidth loads for one set of restricted variables

Assumptions:

1) Each reconstructed frame is stored as two independent fields in memory
2) Luma-chroma data from one field in a macroblock is saved as a storage unit
3) Reconstructed storage units in memory do not cross page boundaries
4)Each reconstructed field memory area starts at a page boundary

Restricted variables:		Worst case Scenario:	
		1) All MBs in a line are coded with bidir. field motion vectors (4 predictors)	
		2) All 4 field predictors come from same memory bank, minimizing overhead	
Picture type (0=>frame):	0	3) OSD information is being updated with page hit	
Motion Estimation mode:	1	4) OSD information is being read for display with page hit	
Compression mode:	2	5) One word of PES header data is read out	
		6) Every set of predictors crosses a page boundary	
		7) Compressed bitstream is being read out with page hit	
		8) Compressed bitstream is being written with page hit	
Tot. MBs in input frame:	8160	OSD read or update cycles per H reset:	39
Frame time (ns):	33333333	Bit buffer read or write cycles per H reset:	39
MBs processed per H_reset (High BW):	8	OSD & VLD read-write cycles per H reset:	228
MBs processed per H_reset (Low BW):	7	PES Header read cycles per H reset:	20
Storage Unit access clocks:	3	Refresh overhead per field:	2048
Storage Units per macroblock:	4		
Storage Units per page:	85	Sum total read & write cycles per H reset (High BW/Wrst Case):	3066
		Sum total read & write cycles per H reset (Low BW/Wrst Case):	2840
		Sum total read & write cycles per H reset (High BW/Best Case):	1962
		Sum total read & write cycles per H reset (Low BW/Best Case):	1874
MB predictor read cycles per H reset (High BW/Wrst Case):	1777	Per field total bandwidth including refresh:	1546673
MB predictor read cycles per H reset (Low BW/Wrst Case):	1557	Memory bus clock frequency:	92800380
Storage Units for display per H reset:	30		
Display decoder read cycles per H reset (High BW):	174	Bandwidth calculations assuming on-chip block-to-scan conversion:	
Display decoder read cycles per H reset (Low BW):	168	Sum total read & write cycles per H reset (High BW/Wrst Case):	2199
Display line buffer read and write:	867	Sum total read & write cycles per H reset (Low BW/Wrst Case):	1973
MB predictor read cycles per H reset (High BW/Best Case):	673	Sum total read & write cycles per H reset (High BW/Best Case):	1095
MB predictor read cycles per H reset (Low BW/Best Case):	591	Sum total read & write cycles per H reset (Low BW/Best Case):	1007
		Per field total bandwidth including refresh:	1058986
		Memory bus clock frequency:	63589160



RCA88130
M. Deiss et al.

FIGURE 1

What is claimed is as follows:

1. A system for processing MPEG compressed image information, said system comprising:

an MPEG decoder capable of receiving an encoded MPEG formatted data stream and providing at the output of said decoder decompressed pixel blocks constituting an image frame;

a compressor for recompressing said decompressed pixel blocks into recompressed pixel blocks;

a frame memory for storing said recompressed pixel blocks;

a first decompressor for decompressing said recompressed pixel blocks for providing motion compensation information to said decoder;

an output network; and

a second decompressor for decompressing said recompressed pixel blocks as needed for said output network.

2. A system according to claim 1, wherein:

said compressor comprises plural data compression networks; and

said first decompressor comprises plural decompression networks.

3. A system according to claim 1, wherein:

said MPEG decoder processes interleaved data blocks.

4. A system according to claim 3, wherein:

said MPEG decoder employs piplined data processing.

Abstract

A television receiver with an MPEG decoder is configurable for full high definition decoding and display, or reduced cost lower definition display. The MPEG decoder (28) uses a controllable dual-mode data reduction network selectively employing horizontal detail reduction (14) and data recompression (16) between the decoder and the decoder frame memory (22) from which image information to be displayed is derived. The amount of data reduction is manufacturer selected in accordance with the resolution of the display device, e.g., equal to or less than high definition resolution. The frame memory size is also manufacturer selected in accordance with the resolution of the display device. The system employs plural compression and decompression networks and pipelined processing of interleaved pixel blocks.